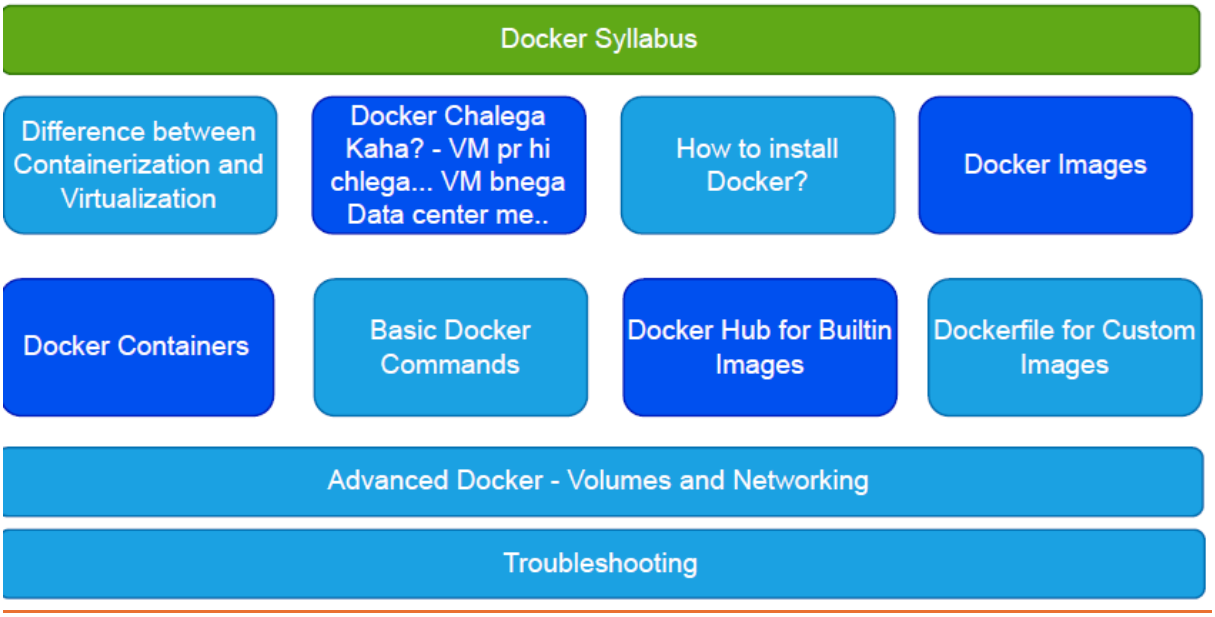
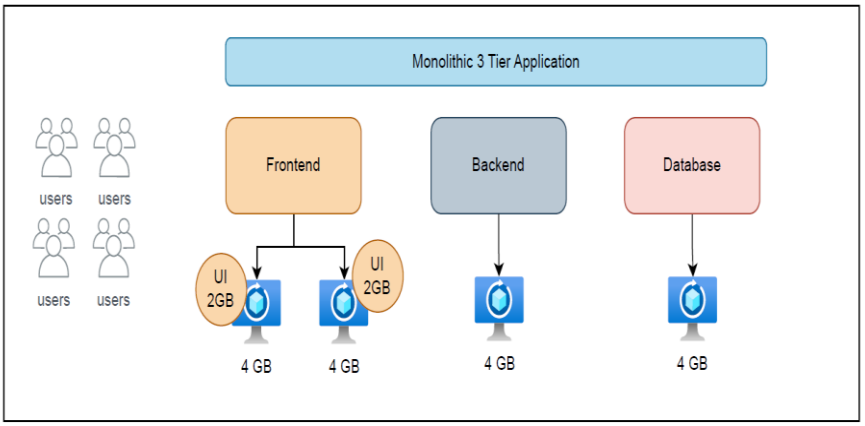
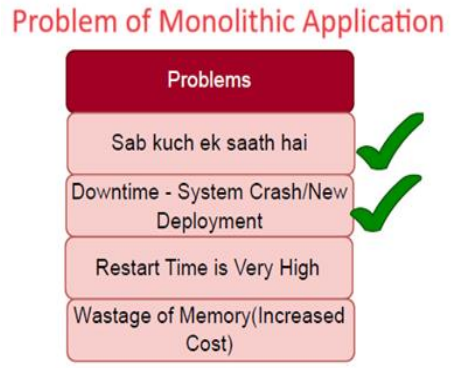
DOCKER



Monolithic Application:





Why we use microservices over monolithic applications?

We use microservices over monolithic applications because microservices break down large applications into smaller, independent parts. This makes it easier to:

1. **Develop**: Teams can work on different parts (microservices) without affecting others.
2. **Scale**: You can scale only the services that need more resources instead of the whole app.
3. **Deploy**: Changes can be made and deployed to individual services without shutting down the entire application.
4. **Maintain**: Debugging and updating smaller services is easier and faster compared to a large,

Virtualization: Virtualization allows us to run multiple operating systems on single physical machine.

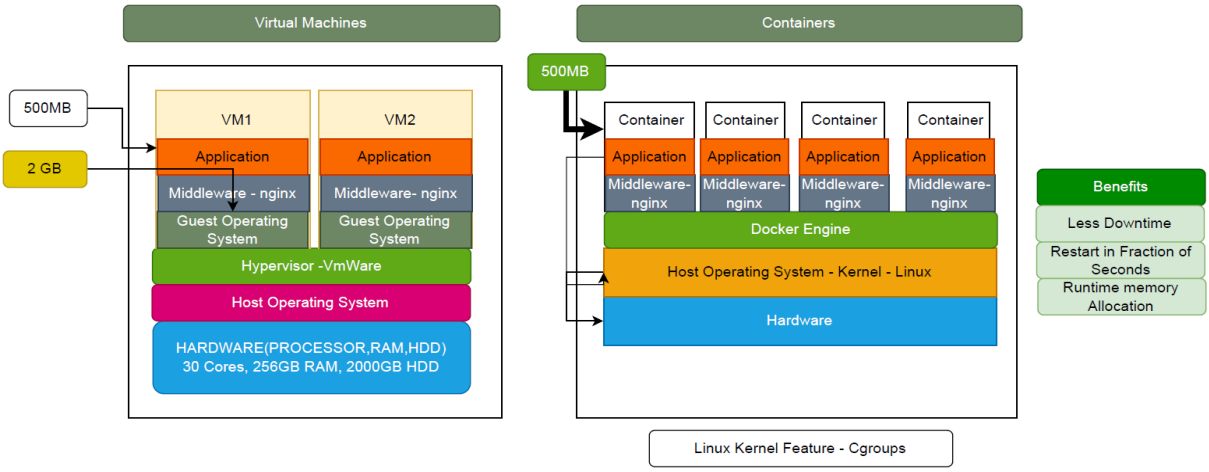
How Virtualization Works: A special software called a hypervisor (like VMware or Hyper-V) sits between the physical hardware and the VMs. It allocates resources (CPU, memory, storage) from the physical machine to each virtual machine

Containerization: Containerization is the process of packaging an application and all its dependencies (libraries, configuration files) into a "container" that can run on any environment.

How Containerization Works: Instead of having a full OS for each app (like in virtualization), containers share the host OS, making them lightweight. A tool like Docker is used to create and manage containers.

Difference between Hypervisor1 & Hypervisor2

Hypervisor 1	Hypervisor2
Directly installed on hardware	Installed on existing OS
High performance, more efficient	Slightly slower due to the overhead of the host OS
Large-scale environments (e.g., data centers)	Personal use or small environments (e.g., testing on a laptop)
VMware ESXi, Hyper-V (bare metal)	VMware Workstation, VirtualBox



Linux feature:

1. Namespaces (Isolation):

Namespaces in Docker ensure that each container runs in its own isolated environment. This means containers don't see or interfere with each other's processes, files, or networks. It's like giving each container its own private view of the system.

- Process Namespace: Containers only see their own processes.
- File System Namespace: Each container has its own file system.
- Network Namespace: Each container has its own network settings (like IP address).
- User Namespace: Each container has separate user IDs, so users inside a container can't access the host system.

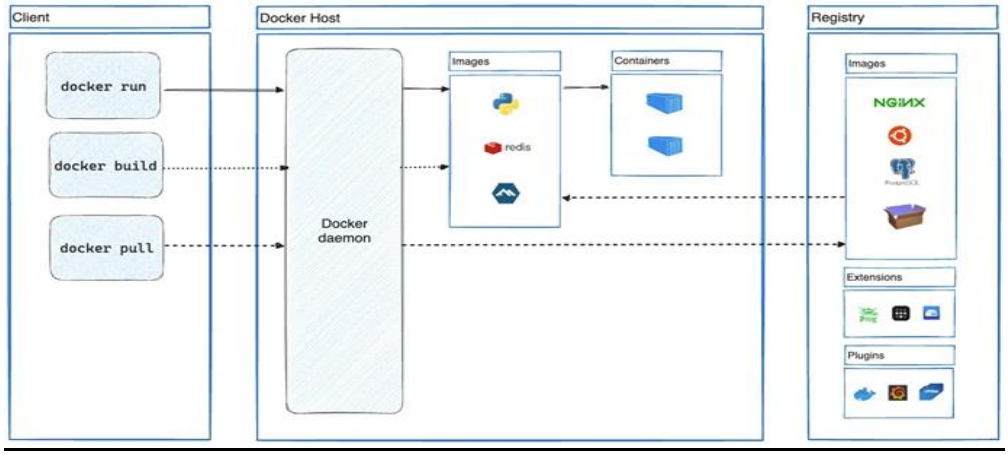
In simple words: Namespaces make sure that each container thinks it's running on its own, separate from others.

2. cgroups (Resource Control):

cgroups (control groups) help Docker manage the resources (CPU, memory, disk) that each container can use. This prevents one container from using up too much of the system's resources and affecting others.

- **CPU Limiting**: You can control how much CPU power a container gets.
- **Memory Limiting**: You can set limits on how much memory a container can use.
- **Disk I/O Limiting**: You can manage how much disk access each container has.
  In simple words: cgroups make sure each container gets only its fair share of system resources, so no container can hog everything.

Architecture of Docker



- **Docker client:** Is the primary way that the docker user interact with.When we give commands to docker client (like run,create container ,pull) it send to docker daemon.
- **Docker Engine:** It takes commands from docker client process them and manages container creation, starting, stopping and other. There are two parts of docker engine:
  1) **Docker daemon**: Runs in background and handles the actual operation like container creation, images and volumes.
  2) **Rest APIs**: Allows others tool or script to communicate with docker daemon
- **Docker Images:** These are like templates that include everything needed for an application to run, such as the code, libraries, and settings. You can think of it as a blueprint for creating containers.
- **Docker Containers:** These are instances of Docker images, like the "live" version of the image. Containers run the actual applications, isolated from the rest of your system, making it easier to manage and deploy applications consistently.
- **Docker Registry:** A place where Docker images are stored and shared. Docker Hub is a popular public registry.

Types of Docker Networks

- **Bridge**: Default network type. Used for standalone containers.
- **Host**: Removes network isolation between the container and the Docker host. Useful for performance.
- **Overlay**: Enables containers across multiple Docker hosts to communicate. Often used in Docker Swarm.
- **Macvlan**: Assigns a MAC address to a container, making it appear as a physical device on the network.
- **None**: Disables all networking for a container. Isolates the container completely from the network. No communication with other containers or the host. Useful for applications that don't require network access. No IP address is assigned. No discovery; containers are completely isolated.

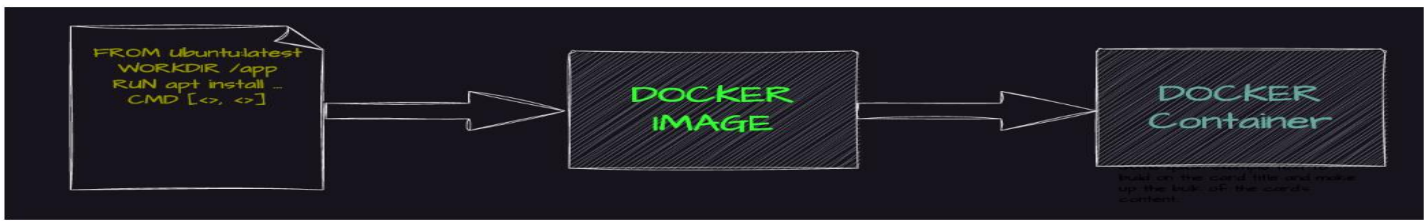
Docker Life Cycle

Docker LifeCycle

We can use the above Image as reference to understand the lifecycle of Docker.

There are three important things,

1. docker build -> builds docker images from Dockerfile
2. docker run -> runs container from docker images
3. docker push -> push the container image to public/private regestries to share the docker images.



Basic Docker Commands

Image related commands

Command	Description
docker images	Lists all Docker images on your system.
docker pull <image-name>	Downloads a Docker image from a registry (e.g., Docker Hub).
docker build -t <image-name> .	Builds a Docker image from a Dockerfile in the current directory, tagging it with `<image-name>`.
docker rmi <image-id>	Deletes a Docker image by its ID.
docker rmi -f <image-id>	Forcefully deletes a Docker image (even if containers are using it).
docker tag <image-id> <new-tag>	Assigns a new tag to an existing image.
docker push <image-name>	Pushes a tagged image to a Docker registry (like Docker Hub).
docker inspect <image-id>	Shows detailed information (metadata, config) about a specific image.
docker history <image-name>	Shows the layer history of an image, detailing the steps used to build it.
docker save -o <file-name>.tar <image-name>	Saves an image to a tar archive file.
docker load -i <file-name>.tar	Loads an image from a tar archive file.
docker image prune	Removes unused/dangling images to free up space.
docker image ls	Lists Docker images (alternative command to `docker images`).
docker search <image-name>	Searches for an image in a Docker registry (e.g., Docker Hub).
docker image inspect <image-name>	Displays detailed information about an image (same as `docker inspect`).

Container Related commands

Command	Description
docker ps	Lists running containers.
docker ps -a	Lists all containers, including stopped ones.
docker run <image>	Creates and starts a new container from an image.
docker run -d <image>	Runs a container in detached mode (in the background).
docker run -it <image-name> bash	Creates and starts a new container interactively with a terminal.
docker exec -it <container-id> bash	Opens an interactive terminal session inside a running container.
docker exec -it <container> <cmd>	Executes a command in a running container interactively.
docker stop <container>	Stops a running container.
docker start <container>	Starts a stopped container.
docker restart <container>	Restarts a running or stopped container.
docker rm <container>	Removes a stopped container.
docker rm -f <container>	Forcefully removes a running container.
docker logs <container>	Displays the logs of a container.
docker inspect <container>	Displays detailed information about a container.
docker attach <container>	Attaches to a running container's console.
docker commit <container> <new_image>	Creates a new image from a container's changes.
docker prune	Removes all stopped containers.
docker cp <container>:<path> <host_path>	Copies files/folders from a container to the host.
docker cp <host_path> <container>:<path>	Copies files/folders from the host to a container.
docker top <container-id>	Displays the running processes inside a container.

Network Related Command

Command	Description
docker network connect (docker network connect my_network my_container)	Connect a container to a network
docker network create (docker network create my_network)	Create a network
docker network disconnect (docker network disconnect my_network my_container)	Disconnect a container from a network
docker network inspect (docker network inspect <network_name>)	Display detailed information on one or more networks
docker network ls	List networks
docker network prune	Remove all unused networks
docker network rm	Remove one or more networks
docker run --network my_network my_image	Run a Container in a Specific Network

Volume Related Command

Command	Description
docker volume create <volume_name>	Create a volume
docker volume inspect <volume_name>	Display detailed information on one or more volumes
docker volume ls	Lists all Docker volumes.
docker volume prune	Removes all unused volumes.
docker volume rm <volume_name>	Remove one or more volumes
docker volume update	Update a volume (cluster volumes only)
docker run -d -v my_volume:/data my_image	Using a Volume with a Container
docker run -v <volume_name>:<container_path> <image>	Mounts a volume into a container.
docker run -v /host/path:/container/path <image>	Runs a container and mounts a directory from the host to the specified path in the container (bind mount).
docker volume create --driver <driver_name> <volume_name>	Creates a volume with a specified driver.
docker volume ls -q	Lists volume names only (quiet mode).

DOCKER Hub Related Command

Command	Description
docker login	Log in to your Docker Hub account.
docker logout	Log out of your Docker Hub account.
docker push <image-name>	Upload your local image to Docker Hub.
docker pull <image-name>	Download an image from Docker Hub to your local machine.
docker search <image-name>	Search for images on Docker Hub.
docker tag <local-image> <repo/name:tag>	Tag a local image to prepare it for pushing to Docker Hub.
docker images	List all local images, including those pulled from Docker Hub.
docker rmi <image-name>	Remove a local image (cannot be used if it's tagged with a Docker Hub repository).

Docker Expose Related Command

Command	Description
EXPOSE <port>	Declares that the container listens on the specified port at runtime (used in Dockerfile).
EXPOSE <port>/<protocol>	Declares a specific port with a protocol (e.g., TCP or UDP) in the Dockerfile.
docker run -p <host-port>:<container-port>	Maps a port on the host to a port on the container, making it accessible outside.
docker ps	Lists running containers and shows which ports are exposed and mapped.
Note : The EXPOSE command is primarily informational and does not actually publish the port; you still need to use -p or --publish when running the container to make the port accessible.	

DOCKER FILE

Instruction	Description		Instruction	Description
ADD	Add local or remote files and directories.		SHELL	Set the default shell of an image.
ARG	Use build-time variables.		VOLUME	Create volume mounts.
CMD	Specify default commands.		WORKDIR	Change working directory.
COPY	Copy files and directories.		HEALTHCHECK	Check a container's health on startup.
ENTRYPOINT	Specify default executable.		LABEL	Add metadata to an image.
ENV	Set environment variables.		MAINTAINER	Specify the author of an image.
EXPOSE	Describe which ports your application is listening on.		ONBUILD	Specify instructions for when the image is used in a build.
FROM	Create a new build stage from a base image.		STOPSIGNAL	Specify the system call signal for exiting a container.
RUN	Execute build commands.		USER	Set user and group ID.

Difference B/W Copy and Add command.

- Use **COPY** when you only need to copy files and directories.
- Use **ADD** when you need to download files from a URL or unpack compressed files but be cautious as it can add unnecessary complexity.

Difference B/W CMD and Entry points command.

CMD

- What it does: Sets default commands or options for a container.
- Overridable: Can be changed when you run the container. If you provide a command in docker run, it will replace CMD.
- Example: CMD ["echo", "Hello, World!"]

ENTRYPOINT

- What it does: Defines the main command that will always run when the container starts.
- Overridable: Less easily overridden. The command specified in docker run will be added as arguments, but the main command remains the same.
- Example: ENTRYPOINT ["python", "app.py"]

CUSTOM FILE

Prerequisite:

1. Docker VM needed
2. File name must be with name of Dockerfile.

Steps

1. Need to create a One directory and inside the directory we need to create dockerfile

```
Last login: Tue Oct  1 07:03:39 2024 from 165.225.124.86
preeti@Docvm:~$ ls
preeti  snap
preeti@Docvm:~$ cd preeti
preeti@Docvm:~/preeti$ ls
Dockerfile
preeti@Docvm:~/preeti$
```

2. vi Dockerfile

3. Example of Dockerfile

```
FROM ubuntu:latest
RUN apt-get update
RUN apt-get install -y nginx
RUN echo "hi this is my first dockefile" > /var/www/html/index.html
#COPY index.html /var/www/html/index.html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"i]
~
~
~
```

- **FROM ubuntu:latest**:

Start with the latest version of Ubuntu as the base image. Image is from Docker hub.

- **RUN apt-get update**:

Update the package list to get the latest available software.

- **RUN apt-get install -y nginx**:

Install Nginx web server without asking for confirmation.

- **RUN echo "hi this is my first dockerfile" > /var/www/html/index.html**:

Create an HTML file with the message in the default web directory for Nginx.

- **EXPOSE 80**:

Indicate that the container will listen on port 80 (HTTP).

- **CMD ["nginx", "-g", "daemon off;"]**:

Start Nginx in the foreground to keep the container running.

4. Now ,we need to create dockerfile

docker buildx build -t tondufile .

```
preeti@Docvm:~/preeti$ sudo docker buildx build -t tondufile .
[+] Building 1.0s (8/8) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 261B
 => [internal] load .dockerignore
 => => transferring context: 2B
```

5. Need to check the above tondufie image

```
Run 'docker image COMMAND --help' for more information on a command.
root@Docvm:/home/preeti/preeti# docker image ls
REPOSITORY         TAG       IMAGE ID       CREATED              SIZE
preetifile         latest    545d879ae8a8   About an hour ago    127MB
tondufile          latest    170e2b3245a4   About an hour ago    127MB
jenkins/jenkins    latest    f21bb75b933f   6 days ago           473MB
wordpress          latest    e4ef5270c81e   2 weeks ago          699MB
nginx              latest    9527c0f683c3   6 weeks ago          188MB
```

6. Need to create container and expose

```
See 'docker run --help'.
preeti@Docvm:~/preeti$ sudo su
root@Docvm:/home/preeti/preeti# docker run -d -p 8080:80 tondufile
20247699f2f3db1481de79667650a13965d531a463d09d2d45184a4cfb2e6851
root@Docvm:/home/preeti/preeti# docker image
```

7. Now browse ip with port

hi this is my first dockefile

MULTISTAGE DOCKERFILE

A multi-stage Dockerfile is a special type of Dockerfile that allows you to create smaller, more efficient Docker images by separating the build process into different stages. Here's a simple breakdown of what that means:

What is a Multi-Stage Dockerfile?

1. **Multiple Stages:**
   - A multi-stage Dockerfile uses multiple FROM statements, which define different stages in the image-building process. Each stage can have its own base image.
2. **Build and Run Separation:**
   - You can use a heavy image (like Ubuntu) to build your application and a lighter image (like Alpine) to run it. This means you can compile code or install dependencies in one stage and then copy only the necessary files to the final image.
3. **Reduced Image Size:**
   - By only including the files you need in the final image, the size of the Docker image is significantly reduced. This makes it faster to download and run.
4. **Cleaner Development:**
   - It keeps your Dockerfile organized by clearly separating the build environment from the runtime environment.

How It Works: Step by Step

1. **Define a Build Stage:**
   - In the first stage, you might use a full operating system (like Ubuntu) to install tools, compile code, or gather files needed for your application.
2. **Copy Files to the Final Stage:**
   - After building, you copy only the necessary files (like compiled binaries or static files) from the first stage to a second stage.
3. **Run the Application:**
   - The second stage uses a lightweight image (like Nginx or Alpine) to run the application, which has everything needed to execute the code without extra tools or libraries.

Example:

```
FROM ubuntu:latest AS tondu
RUN apt-get update
RUN apt-get install -y nginx
RUN echo "hi this is my multistage dockerfile" > /var/www/html/index.html


FROM nginx:alpine
COPY --from=tondu /var/www/html/index.html /usr/share/nginx/html/index.html
EXPOSE 80
```

Dockerfile Breakdown

Stage 1 (Build Stage): Build the Application: The first stage prepares everything needed for the application. # Stage 1: Build the application

FROM ubuntu:latest AS tondu

RUN apt-get update

RUN apt-get install -y nginx

RUN echo "hi this is my multistage dockerfile" > /var/www/html/index.html

1. **FROM ubuntu:latest AS tondu:**
    - This line specifies the base image for the first stage of the build. It uses the latest version of the Ubuntu operating system and labels this stage as tondu. The name tondu is arbitrary and can be changed as desired.
2. **RUN apt-get update:**
    - This command updates the package list in the Ubuntu image, ensuring that the latest package versions are available for installation. This step is essential for installing software packages.
3. **RUN apt-get install -y nginx:**
    - This command installs the Nginx web server. The -y flag automatically confirms any prompts that may arise during the installation, allowing the process to continue without manual intervention.
4. **RUN echo "hi this is my multistage dockerfile" > /var/www/html/index.html:**
    - This line creates an HTML file named index.html in the directory /var/www/html. It contains the text "hi this is my multistage dockerfile." This file will be served by the Nginx server.

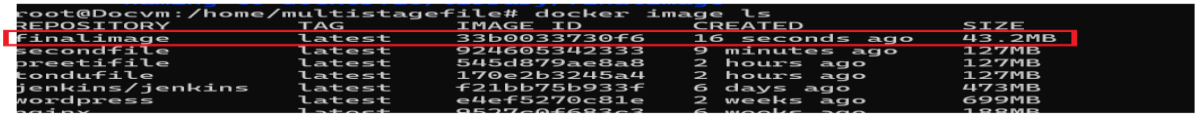
Stage 2 (Application Stage): Run the Application: The second stage sets up the runtime environment, providing a clean and efficient image for serving the web page.

FROM nginx:alpine

COPY --from=tondu /var/www/html/index.html /usr/share/nginx/html/index.html

EXPOSE 80

1. **FROM nginx:alpine:**
    - This line starts the second stage of the Dockerfile. It uses a lightweight version of Nginx based on Alpine Linux, which is more efficient for running applications due to its small size.
2. **COPY --from=tondu /var/www/html/index.html /usr/share/nginx/html/index.html:**
    - This command copies the index.html file that was created in the first stage (tondu) into the appropriate directory for serving web content in the Nginx container. The destination path /usr/share/nginx/html/index.html is where Nginx looks for HTML files to serve.
3. **EXPOSE 80:**
    - This instruction informs Docker that the container listens on port 80. It does not publish the port but serves as documentation. When you run the container, you would need to map this port to a port on your host machine to access the web server.



Docker Registry:

A Docker Registry is a place where Docker images are stored and managed. Here's a simple explanation:

What is a Docker Registry?

- **Storage for Images:** Think of it like a warehouse for Docker images. It holds all the images you create or use.
- **Image Management**: It helps you organize, version, and distribute Docker images, making it easier to manage your applications.
- **Public or Private:** A Docker Registry can be public (like Docker Hub) or private (for your own use). You can set it up on your own servers if you want.
- **Local or Remote:** You can have a registry that's local (on your machine) or remote (accessible over the internet).

Why Use a Docker Registry?

- **Efficient Sharing:** It allows you and your team to share images easily.
- **Control:** You have control over your images, especially if you set up a private registry.
- **Security:** You can keep sensitive images in a private registry to protect them.

In summary, a Docker Registry is essential for storing and managing Docker images, whether for personal projects or team collaboration

Docker Hub:

Docker Hub is a cloud-based service where you can store and share Docker images. Here's a simple breakdown:

What is Docker Hub?

- **Cloud Storage**: Think of it like a library for Docker images. You can keep your images there and access them from anywhere.
- **Image Sharing**: You can share your Docker images with others. This makes it easy for teams to collaborate on projects.
- **Pre-Built Images**: Docker Hub has many pre-built images (like software applications) that you can use right away, saving you time.
- **Version Control**: You can keep track of different versions of your images, making it easy to roll back if needed.
- **Free and Paid Options**: Docker Hub offers both free public repositories and paid private repositories for sensitive projects.

Why Use Docker Hub?

- **Convenience**: Easily store and retrieve images.
- **Collaboration**: Work with others by sharing images.
- **Easy Access**: Pull images from Docker Hub to your local machine with a simple command.

In short, Docker Hub makes it easy to manage and share Docker images, helping developers work more efficiently.

MULTISTAGE DOCKERFILE DEEP DRIVE

Java Dockerfile

```
FROM openjdk:11-jre-slim
WORKDIR /app
COPY target/myapp.jar myapp.jar
ENTRYPOINT ["java", "-jar", "myapp.jar"]
```

React Dockerfile

```
# React Dockerfile
FROM node:14 as build
WORKDIR /app
COPY package.json ./
COPY package-lock.json ./
RUN npm install
COPY . .
RUN npm run build
FROM nginx:alpine
COPY --from=build /app/build /usr/share/nginx/html
CMD ["nginx", "-g", "daemon off;"]
```

Python Dockerfile

```
FROM python:3.9
WORKDIR /app
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["flask", "run", "--host=0.0.0.0"]
```

```
/project-root
|-- /java-app
|   |-- Dockerfile
|   |-- target/myapp.jar
|
|-- /react-app
|   |-- Dockerfile
|   |-- package.json
|   |-- package-lock.json
|   |-- src/
|
|-- /python-app
|   |-- Dockerfile
|   |-- requirements.txt
|   |-- app.py
|
|-- /mysql-app
|   |-- Dockerfile
|
|-- /php-app
|   |-- Dockerfile
|   |-- index.php
|
|-- nginx.conf
|-- docker-compose.yml
```

MySQL Dockerfile

```
# MySQL Dockerfile (use official image)
FROM mysql:5.7
ENV MYSQL_DATABASE=mydb
ENV MYSQL_USER=user
ENV MYSQL_PASSWORD=password
ENV MYSQL_ROOT_PASSWORD=rootpassword
```

PHP Dockerfile

```
# PHP Dockerfile
FROM php:7.4-apache
WORKDIR /var/www/html
COPY . .
```

Dotnet Docker file

```
# Build stage
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /app
COPY *.csproj ./
RUN dotnet restore
COPY . .
RUN dotnet publish -c Release -o out
# Final stage
FROM mcr.microsoft.com/dotnet/aspnet:5.0
WORKDIR /app
COPY --from=build /app/out .
ENTRYPOINT ["dotnet", "YourApp.dll"]
```

Step Wise Implementation:

Note: All application file kept in remote repo like GitHub or Azure Repo, So we proceed Step By Step.

Step1: login in docker server, make one application folder.

Step2: Now Inside application folder, clone application code in our docker server.

```
satish@satish:~/javaapp$ git clone https://github.com/devopsinsiders/JavaLoginPracticeApp.git
Cloning into 'JavaLoginPracticeApp'...
remote: Enumerating objects: 54, done.
remote: Counting objects: 100% (54/54), done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 54 (delta 12), reused 46 (delta 9), pack-reused 0 (from 0)
Unpacking objects: 100% (54/54), 1007.10 KiB | 1.24 MiB/s, done.
satish@satish:~/javaapp$ ls
JavaLoginPracticeApp
satish@satish:~/javaapp$ ls
JavaLoginPracticeApp
satish@satish:~/javaapp$ cd JavaLoginPracticeApp/
```

Step3: Now go inside application folder “”JavaLoginPractiseApp”, Need to create one Dockerfile for write Dockerfile code:

```
satish@satish:~/javaapp/JavaLoginPracticeApp$ ll
total 152
drwxrwxr-x 4 satish satish     4096 Oct 12 04:28 ./
drwxrwxr-x 3 satish satish     4096 Oct 12 03:39 ../
-rw-rw-r-- 1 satish satish        0 Oct 12 04:28 Dockerfile
drwxrwxr-x 8 satish satish     4096 Oct 12 03:40 .git/
-rw-rw-r-- 1 satish satish      290 Oct 12 03:40 .gitignore
-rw-rw-r-- 1 satish satish   112733 Oct 12 03:40 image.png
-rw-rw-r-- 1 satish satish    11357 Oct 12 03:40 LICENSE
-rw-rw-r-- 1 satish satish      905 Oct 12 03:40 pom.xml
-rw-rw-r-- 1 satish satish     3521 Oct 12 03:40 README.md
drwxrwxr-x 3 satish satish     4096 Oct 12 03:40 src/
satish@satish:~/javaapp/JavaLoginPracticeApp$
```

Step4: Prerequisite for all language application

Prerequisite	
1	**JAVA BASE APPLICATION (pod.xml) - Maven**
Stage1 (Build)	a. Need to take maven image for build the application
	b. Need to set the working directory inside the container, and go inside it (workdir)
	c. Copy the pod.xml and src file inside the container
	d. Now build the application
	e. Now one war artifact file generated which will use in final stage
Stage2 (Final)	a. Now install webserver to deploy application (tomcat)
	b. Copy the war file into webserver host directory
	c. Expose the application
	d. Start tomcat

2	**REACT APPLICATION (package*.json) - npm**
Stage1 (Build)	a. Need to take npm image for build the application
	b. Need to set the working directory inside the container, and go inside it (workdir)
	c. Copy the package*.json and src file inside the container
	d. Now build the application
	e. Now one build artifact file generated which will use in final stage
Stage2 (Final)	f. Now install nginx webserver to deploy application
	g. Copy the build file into webserver host directory /usr/share/nginx/html
	h. Expose the application

3	**Python APPLICATION (requirement.txt)**
Single Stage	a. Need to take python image for build the application
	b. Need to set the working directory, and go inside it (workdir)
	c. Copy the application file into the container which will show once will create docker image and after deployed this image into container, then will go inside the container and we can check this file
	d. Now we can expose the application

4	**MySQL DB**
Single Stage	a. Take mysql image
	b. Set environment (DB, Username, Password, root password)

5	**Dotnet APPLICATION (*.csproj) - sdk**
Stage1 (Build)	a. Need to take /dotnet/sdk:5.0 image for build the application
	b. Need to set the working directory inside the container, and go inside it (workdir)
	c. Copy the package *.csproj and src file inside the container
	d. Now build the application
	e. Now one build artifact file generated which will use in final stage
Stage2 (Final)	f. Now install aspnet webserver to deploy application
	g. Create workdir /app
	h. Copy the build file into /app --from=build /app/out
	i. Expose the application

Step5: Now for Java Application, Need to use below docker image

Cd /javaapp/javaloginpractiseapp/

Vi Dockerfile

```
# Build stage
FROM maven:3.8.1-openjdk-11 AS build
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package

# Final stage
FROM tomcat:9.0
COPY --from=build /app/target/* /usr/local/tomcat/webapps/
EXPOSE 8080
CMD ["catalina.sh", "run"]
```

Step6: Now run docker build command for generated customer image

Docker buildx build -t <name of customer image>

Docker buildx build -t javaapp2 .

```
satish@satish:~/javaapp/JavaLoginPracticeApp$ vi Dockerfile
satish@satish:~/javaapp/JavaLoginPracticeApp$ docker buildx build -t javaapp2 .
[+] Building 2.1s (15/15) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 284B
 => [internal] load metadata for docker.io/library/tomcat:9.0
 => [internal] load metadata for docker.io/library/maven:3.8.1-openjdk-11
 => [auth] library/tomcat:pull token for registry-1.docker.io
 => [auth] library/maven:pull token for registry-1.docker.io
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [stage-1 1/2] FROM docker.io/library/tomcat:9.0@sha256:189528c1103b212ca44ca608d110dd07cfb57a6d67fc5c27d2a578f97d2ab607
 => [internal] load build context
 => => transferring context: 1.67kB
 => [build 1/5] FROM docker.io/library/maven:3.8.1-openjdk-11@sha256:aaf506d47cd2ec8f62fc1ff74065eda5614738e8ea61bad9b32da0360b9498cd
 => CACHED [build 2/5] WORKDIR /app
 => CACHED [build 3/5] COPY pom.xml .
 => CACHED [build 4/5] COPY src ./src
 => CACHED [build 5/5] RUN mvn clean package
 => CACHED [stage-1 2/2] COPY --from=build /app/target/* /usr/local/tomcat/webapps/
 => exporting to image
 => => exporting layers
 => => writing image sha256:1dc89238d0e54f181ee16de9651e47f19bdc1e490f091f520922e3170d3f5d49
 => => naming to docker.io/library/javaapp2
satish@satish:~/javaapp/JavaLoginPracticeApp$ docker run -p 8080:8080 --name=javaapptest3 javaapp2:latest
```

Step7: Now Run container based on these image:

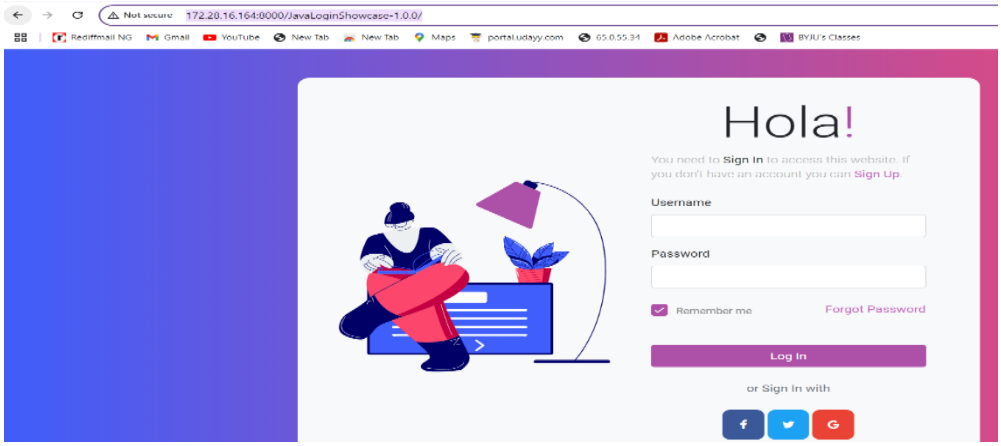
Docker run -d -p <hostport>:<containerport> imagename

Docker run -d -p 8000:8080 javaapp2

```
satish@satish:~/javaapp/JavaLoginPracticeApp$ docker run -d -p 8000:8080 javaapp2
0fa2ed9d2aa57baecffd5af5383f7f3933e2c59ba082c7dc10f45ab94c3eb0ff
```

Step8: **Now able to run application by using hostip and port no**

http://172.28.16.164:8000/JavaLoginShowcase-1.0.0/



Step9: **Now upload this image to our registry**

Docker login

Docker image ls

```
satish@satish:~/javaapp/JavaLoginPracticeApp$ docker login
Authenticating with existing credentials...
WARNING! Your password will be stored unencrypted in /home/satish/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credential-stores

Login Succeeded
satish@satish:~/javaapp/JavaLoginPracticeApp$ docker image ls
REPOSITORY                      TAG       IMAGE ID       CREATED          SIZE
javaapp                         latest    b0085a4777db   51 minutes ago   470MB
javaapp2                        latest    1dc89238d0e5   51 minutes ago   470MB
gcr.io/k8s-minikube/kicbase     v0.0.45   aeed0e1d4642   5 weeks ago      1.28GB
sonarqube                       latest    2433ac783140   2 months ago     1.07GB
```

Docker tag javaapp satishranjan7183/javaapp2:latest

```
satish@satish:~/javaapp/JavaLoginPracticeApp$ docker tag javaapp satishranjan7183/javaapp2:latest
satish@satish:~/javaapp/JavaLoginPracticeApp$ docker push satishranjan7183/javaapp2:latest
The push refers to repository [docker.io/satishranjan7183/javaapp2]
b019fab21fed: Pushed
5f70bf18a086: Mounted from library/tomcat
1623cee902e6: Mounted from library/tomcat
5a433f41592a: Mounted from library/tomcat
fa2b783d3fa5: Mounted from library/tomcat
c563ba6d77b2: Mounted from library/tomcat
bbf97972c239: Mounted from library/tomcat
5f62e0ab37f7: Mounted from library/tomcat
fa0f10cc481e: Mounted from library/tomcat
latest: digest: sha256:1b47bc779e46716bd8ef90bdb3a673ff4f6fc00b3b6624eb41a1ff9224ee99b5 size: 2412
satish@satish:~/javaapp/JavaLoginPracticeApp$
```

Docker push satishranjan7183/javaapp2:latest