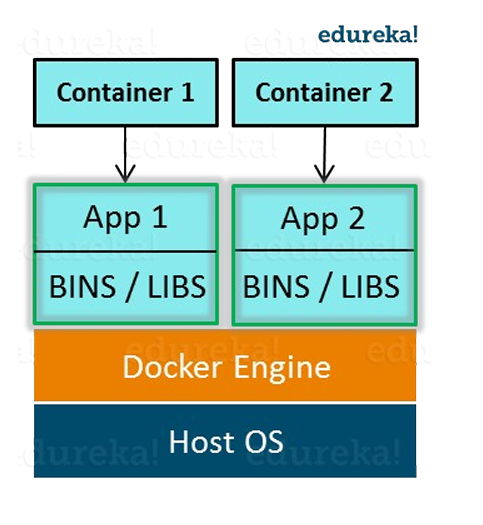
**What is docker?**

Let me give you an introduction to Docker first. Docker is a containerization platform that packages your application and all its dependencies together in the form of Containers to ensure that your application works seamlessly in any environment.



As a developer, I can build a container which has different applications installed on it and give it to my QA team who will only need to run the container to replicate the developer environment.

**Key concepts:**

* **Containers**: Lightweight, standalone, and executable packages
* **Images**: Read-only templates used to create containers. They contain the application and its dependencies. You can think of images as the blueprint for containers.
* **Docker file**: A text file containing a series of instructions on how to build a Docker image. It includes commands for installing software, copying files, setting environment variables, and more.
* **Docker Engine**: The runtime that manages containers on your system. It includes both the Docker Daemon (which runs in the background and handles container operations) and the Docker CLI (command-line interface).
* **Docker Hub**: A cloud-based repository for sharing Docker images. It’s a central place where you can find and upload images.

**How Docker Works:**

1. **Write a docker file**: Define your application’s environment in a docker file. Specify the base image (e.g., node, python), and include instructions for installing necessary packages and copying your application code.
2. **Build an Image**: Use the Docker CLI to build an image from your docker file. The command docker build -t myapp . creates an image tagged myapp.
3. **Run a Container**: Start a container using the image you built.

For example: docker run -d --name mycontainer -p 80:80 myapp runs a container in detached mode, name of the container and maps port 80 of the container to port 80 on the host.

**Manage Containers**: Use Docker commands to start, stop, restart, and remove containers.

**IMAGES**:

* 1. docker push <image\_name> -Share your images by pushing them to Docker Hub
  2. docker pull <image\_name> - You can also pull images from Docker Hub
  3. **Write a docker file [ all docker files given below ]**
  4. From the docker file we **build an Image**
  5. **docker build -t myapp .** – execute docker file and create docker image from docker file. creates an image tagged myapp.
  6. docker images – list all images
  7. docker rmi <image name> - remove docker images.

Containerization is a method of packaging an application and its dependencies into a standardized unit called a container. This approach helps ensure that applications run consistently across various computing environments.

Why use containers:

1. Isolation
2. Resource Efficiency
3. Scalability and Flexibility
4. Faster Deployment

Use cases:

1. Development and Testing
2. Continuous Integration/Continuous Deployment (CI/CD)

**CONTAINERS:**

1. docker run -d --name mycontainer -p 80:80 mynodeimage [ container name- mycontainer, 80- ec2 port no: , 80: web browser port no: , already created image name- mynodeimage]
2. docker ps - lists running containers
3. docker ps -a - list all running and stopped container
4. docker stop <container\_id> - stops a container
5. docker exec - to run commands inside a running Docker container. It allows you to interact with a container that’s already up and running, which is useful for debugging, maintenance, or administrative tasks
6. docker inspect – details of images and containers

**VOLUMES:**

* 1. docker volume create my-vol – create a volume
  2. docker volume ls – list all volumes
  3. docker pull mysql – pull mysql image in dockerhub
  4. docker run --name mysqlcontainer -d -e MYSQL\_ROOT\_PASSWORD =admin -v my-vol:/var/lib/mysql mysql

- mysqlcontainer 🡪container name

- MYSQL\_ROOT\_PASSWORD 🡪while using mysql image , we must give an environmental variable . admin [ password we have to set] This sets the root password for MySQL

- my-vol 🡪 already created volume name

- -v my-vol:/var/lib/mysql: Binds the volume my-vol to /var/lib/mysql in the container, which is where MySQL stores its data.

- mysql: Specifies the image to use. By default, Docker will pull the latest MySQL image from Docker Hub.

1. docker volume rm volume1 volume2 volume3 – remove multiple volumes

Developer gives

1. .json file 🡪it denotes write docker file for node.js
2. .txt file 🡪 it denotes write docker file for python
3. .jar file 🡪 it denotes write docker file for java
4. .xml file 🡪 it denotes write docker file for java.

🡪 But developer does not give .jar file. We can create .jar file using build tool [maven /gradel ]

1. Maven [pom.xml]
2. Gradel [build.gradle]

|  |  |  |
| --- | --- | --- |
| File format | Docker file | Package management tool |
| package. json | Node.js | Npm [Node Package Manager] |
| requirement.txt | python | pip |
| app.jar  pom.xml | java | Maven [pom.xml]  Gradel [build.gradle] |

Suppose package.json file [ default file name] not given in github repo, it means it’s an already build application. For run that application, we need web server [ nginx or Apache ]

|  |  |  |
| --- | --- | --- |
| APPLICATION | EXPOSE | CMD/ENTRYPOINT |
| python | 5000 (our choice) | ["python" , "app.py"] |
| java | 8000 (our choice) | ["java" , "-jar" , "app.jar"] |
| maven | - | - |
| Nodejs (nginx webserver) | 80 (must be 80) | ["nginx", "-g", "daemon off;"] |
| Nodejs | 5000 (our choice) | [“node” , “app.js”] |

app.py - file name. Got that file from developer

app.jar -jar file name

web root [where website files are served from]

1. Apache - /var/www/html/
2. Nginx - /usr/share/nginx/html

CMD – CMD instruction specifies the default command that will be executed when a container is started from the built image.

**docker file: [ nginx]**

FROM nginx:latest

WORKDIR /usr/share/nginx/html

COPY build/ . (or) COPY . . [hint: in given link, we have build folder, so gave build. Everything in build file can copied in docker image]

EXPOSE 80

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

**docker file: [node]**

FROM node: latest

WORKDIR /app

COPY package.json ./

RUN npm install

COPY . .

EXPOSE 80

CMD [“node” , “app.json”]

**docker file: [ python]**

FROM python:latest

WORKDIR /app

COPY . . (OR) COPY requirements.txt .

RUN pip install -r requirements.txt .

COPY . .

EXPOSE 5000

CMD [“python” , “app.py”]

**docker file: [ java]**

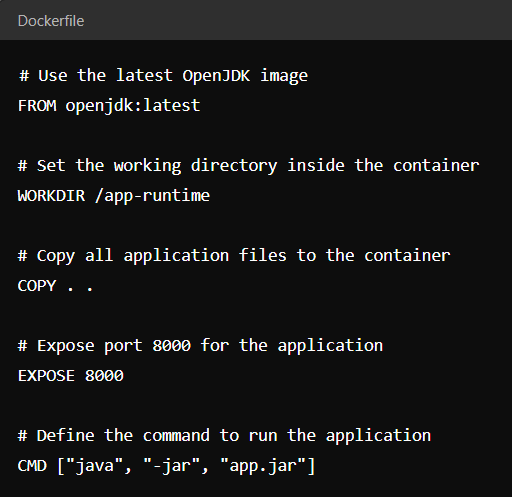
FROM openjdk: latest

WORKDIR /app-runtime

COPY . .

EXPOSE 8000

CMD [“java” , “-jar” “app.jar”]



**Docker compose:**

version: '3'

services:

webcontainer:

image: projectimage

ports:

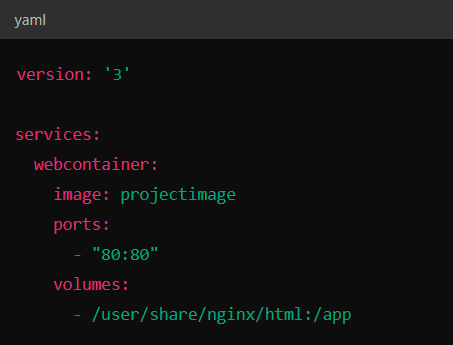
- "80:80"

volumes:

* /user/share/nginx/html:/build

The volumes key should be a list where each item is a mapping between the host path and the container path. The format is:





**Build:**

#!/bin/bash

#build the docker image

docker build -t projectimage .

**deploy:**

#!/bin/bash

docker login -u sharmi2504 -p dckr\_pat\_9SQ5F6VWmpfq\_5dLNHHeqxN2XZI

if [ $GIT\_BRANCH = "dev" ]; then

# Build your project

sh 'chmod +x build.sh'

sh './build.sh'

docker tag projectimage sharmi2504/dev

docker push sharmi2504/dev

elif [ $GIT\_BRANCH = "master" ]; then

sh 'chmod +x build.sh'

sh './build.sh'

docker tag projectimage sharmi2504/prod

docker push sharmi2504/prod

fi

**Jenkins file:**

pipeline {

agent any

stages {

stage('changing file permission') {

steps {

sh 'chmod +x build.sh'

sh 'chmod +x deploy.sh'

}

}

stage('Build') {

steps {

script {

// Build Docker image using build script file

sh './build.sh'

}

}

}

stage('Login') {

steps {

withCredentials([usernamePassword(credentialsId: 'docker-password-id', passwordVariable: 'DOCKER\_PASSWORD', usernameVariable: 'DOCKER\_USERNAME')]) {

sh 'echo $DOCKER\_PASSWORD | docker login -u $DOCKER\_USERNAME --password-stdin'

}

}

}

stage('Deploy') {

steps {

script {

sh './deploy.sh'

}

}

}

}

}