==========
Software Application Architecture
==========

1) Frontend : User Interface (UI)

2) Backend : Business logic

3) Database : Storage

==========
Tech Stack of Application
==========

Frontend : Angular 18v

Backend : Java 17v

Database : MySQL

Note: If we want to run our application code, then we need to setup all required dependencies/softwares in the machine.

Note: dependencies nothing but the softwares which are required to run our application.

==========
Application Environments
==========

=> In realtime we will use several environments to test our application..

1) DEV => 10 machines

2) SIT => 20 machines

3) UAT => 50 machines

4) PILOT => 100 machines

5) PROD (final delivery) => 200 machines

=> DEV env used by developers for code integration testing.

=> SIT env used by Testers for system integration testing.

=> UAT env used by client side team for acceptance testing (Go or No Go)

=> Pilot env used for pre-production testing

=> Prod env used for live deployment (end users can access our app)

=> DevOps team is responsible to setup infrastructure to run our application.

=> We need to install all required softwares (dependencies) to run our application

Note: We need to setup dependencies in all environments to run our application.

Note: There is a chance of doing mistakes in dependencies installation process (version compatability

issues can occur)

==========
Life without Docker
==========

(1) "It works on my machine" problems"

- Developers would build and test apps on their laptops.

- When they move the app to servers, it often breaks because of differences in OS, libraries, versions, configurations, etc.

Without Docker, teams would spend hours or days debugging these environment differences.

(2) Complex Software Installations

- installing something like a database (e.g., PostgreSQL) required manually:

    - download right version of s/w
    - install dependencies
    - configure services properly

(3) Heavy Virtual Machines (VMs)

- Before Docker, developers used Virtual Machines like VirtualBox, VMware, or Hyper-V to simulate servers.

- VMs are large, slow to start, and resource-hungry (each VM has a full OS inside).

********** To resolve above probelms we can use Docker **********

==========
What is Docker ?
==========

=> Docker is a free & open source software

=> Docker is used for containerization

Container = Execute application as a package (code + required s/w)

=> With the help of docker, we can run our application in any machine very easily.

=> Docker will take care of dependencies installation required for app execution.

=> We can make our application portable using Docker.

==========
Docker Architecture
==========

1) Dockerfile

2) Docker Image

3) Docker Registry

4) Docker Container

=> Dockerfile is used to specify where is our app-code and what dependencies (softwares) are required for our application execution.

=> Docker Image is a package which contains (app_code + dependencies)

Note: Dockerfile is required to build docker image.

=> Docker Registry is used to store Docker Images.

Note: When we run docker image then Docker container will be created. Docker container is a linux virtual machine.

=> Docker Container is used to run our application.

==========
Docker Setup in Linux VM
==========

@@ Git repo for steps : https://github.com/ashokitschool/DevOps-Documents/blob/main/02-Docker-Setup.md

==========
Docker Commands
==========

docker images : To display docker images available in our system.

docker pull : To download docker image from docker hub

```
$ docker pull <image-name/image-id>
```

docker run : To create docker container based on docker image

```
$ docker run <image-name/image-id>
```

docker ps : to display running docker containers

docker ps -a : To display running + stopped docker containers.

docker ps -q : To display running containers ids.

docker ps -a -f status=exited -q : To display stopped containers ids.

docker stop : To stop running container

```
$ docker stop <container-id>
```

docker rm : To delete stopped container

```
$ docker rm <container-id>
```

docker start : To start the containers which is in stopped state.

```
$ docker start <container-id>
```

docker rmi : To delete docker image

```
$ docker rmi <image-name/image-id>
```

docker logs : To display container logs

```
$ docker logs <container-id>
```

docker system prune : delete stopped containers + un-used images

```
$ docker system prune -a
```

==========
Running Real-world applications using docker images
==========

public docker image name (java springboot app) : ashokit/spring-boot-rest-api

```
$ docker pull ashokit/spring-boot-rest-api

$ docker run ashokit/spring-boot-rest-api
```

Ex : Ex: docker run -p host-port:container-port <image-name>

```
$ docker run -p 9090:9090 ashokit/spring-boot-rest-api
$ docker run -p 9091:9090 ashokit/spring-boot-rest-api
$ docker run -p 9092:9090 ashokit/spring-boot-rest-api
```

Note: Host port number we need to enable in ec2-vm security group inbound rules to allow the traffic.

Note: To access application running in the container we will use below URL

@@@ SB App URL : http://host-public-ip:host-port/welcome/{name}

public docker image name (python app) : ashokit/python-flask-app

```
$ docker run -d -p 5000:5000 ashokit/python-flask-app
```

Note: Host port number we need to enable in ec2-vm security group inbound rules to allow the traffic.

@@@ Python App URL : http://host-public-ip:host-port/

==========
What is Port Mapping ?
==========

Note: By default, services running inside Docker containers are isolated and not accessible from outside.

=> Docker port mapping is the process of linking container port to host machine port.

=> It is used to allow external access to applications running inside the container.

Syntax : docker run -p <host_port>:<container_port> image_name

Note: host port and container port no need to be same.

==========
What is detached mode ?
==========

=> Detached mode in Docker means running a container in the background rather than attaching to its input/output in the terminal.

=> When you run a container in detached mode, Docker starts it and then immediately gives you back control of the terminal. The container continues running in the background.

```
$ docker run -d -p host-port:container-port <image-name>
```

==========

Dockerfile
==========

=> Dockerfile contains set of instructions to build docker image.

=> Inside Dockerfile we will specify below things

1) Where is Application Packaged File (jar or war)

2) Softwares Required To run the application

3) Application Execution Process

=> To write dockerfile we will use below keywords

1) FROM
2) MAINTAINER
3) RUN
4) CMD
5) ENTRYPOINT
6) COPY
7) ADD
8) WORKDIR
9) EXPOSE

====
FROM
=====

=> It is used to specify base image required to run our application.

Ex:

FROM openjdk:17

FROM mysql:8.5

FROM node:19.5

FROM python:3.3

============
MAINTAINER
============

=> MAINTAINER is used to specify who is author of this Dockerfile.

=> This is Optional in Dockerfile.

Ex : MAINTAINER Ashok <ashok.b@oracle.com>

=====
RUN
=====

=> RUN keyword is used to specify instructions (commands) to execute at the time of docker image creation.

Ex:

RUN 'git clone <repo-url>'

RUN 'mvn clean package'

Note: We can specify multiple RUN instructions in Dockerfile and all those will execute in sequential

manner.

========
CMD
========

=> CMD keyword is used to specify instructions (commands) which are required to execute at the time of docker container creation.

Note: CMD is used to run the application inside container.

Ex:

CMD 'java -jar app.jar'

CMD 'python app.py'

Note: If we write multiple CMD instructions in dockerfile, docker will execute only last CMD instruction.

===========
ENTRYPOINT
===========

=> It is used to execute instruction when container is getting created.

Note: ENTRYPOINT is used as alternate for 'CMD' instructions.

CMD "java -jar app.jar"

ENTRYPOINT ["java" "-jar" "app.jar"]

Note-1 : CMD instructions we can override while creating docker container using command line args.

Note-2 : ENTRYPOINT instructions we can't override.

=====
COPY
=====

=> COPY instruction is used to copy the files from source to destination.

Note: It is used to copy application code from host machine to container machine.

Ex:

COPY <source> <destination>

COPY target/app.jar /usr/app/

COPY target/app.war /usr/bin/tomcat/webapps/

=====
ADD
=====

=> ADD instruction is used to copy the files from source to destination.

Ex :

ADD <source> <destination>

ADD <URL> <destination>

========
WORKDIR
========

=> WORKDIR instruction is used to set / change working directory in container machine.

Ex:

COPY target/sbapp.jar /usr/app/

WORKDIR /usr/app/

CMD 'java -jar sbapp.jar'

========
EXPOSE
========

=> EXPOSE instruction is used to specify application is running on which PORT number.

Ex :

EXPOSE 8080

Note: By using EXPOSE keyword we can't change application port number. It is just to provide information to the people who are reading our Dockerfile.

Note: Specifing Expose keyword in Dockerfile is optional.

==
Dockerizing Java Spring Boot Application
==

=> Every JAVA SpringBoot application will be packaged as "jar" file only.

Note: To package java application we will use 'Maven' as a build tool.

=> To run spring boot application we need to execute jar file.

        Syntax : java -jar <jar-file-name>

Note: When we run springboot application jar file then springboot will start tomcat server with 8080 port number (embedded tomcat server).

```
FROM openjdk:17

COPY target/spring-boot-docker-app.jar /usr/app/

WORKDIR /usr/app/

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "spring-boot-docker-app.jar"]

```

==

@@ *SpringBoot with Docker Video* : https://www.youtube.com/watch?v=iGz0cFwt5vI

==

Step-1 : Connect with Docker Host Machine

Step-2 : Install Git and Clone git repo

```
$ sudo yum install git -y
$ git --version
$ git clone https://github.com/ashokitschool/spring-boot-docker-app.git
```

Step-2 : Install Maven and Package project

```
$ sudo yum install maven -y
$ mvn -version
$ cd spring-boot-docker-app
$ mvn clean package
$ ls -l target
```

Step-3 : Create Docker Image (Dockerfile already present in git repo)

```
$ docker build -t <docker-hub-acc-uname>/<image-name>:tagname .
$ docker images
```

Step-4 : Create Docker Container

```
$ docker run -d -p 8080:8080 <image-name>
$ docker ps
$ docker logs <container-id>
```

Step-5 : Enabled host port (8080) in security group inbound rules and access the application

URL : http://public-ip:8080/

==========
how to push docker image to docker hub ?
==========

login into your docker hub account

```
$ docker login
```

push image to docker hub account

```
$ docker push <docker-hub-acc-uname>/<image-name>:tagname
```

==========
Docker Compose
==========

=> Earlier ppl developed projects using Monolithic Architecture (everthing in single app)

=> Now a days projects are developing based on Microservices architecture.

=> Microservices means multiple backend apis will be avialable

Ex:
hotels-api
flights-api
trains-api
cabs-api...

=> For every API we need to create seperate container.

Note: When we have multiple containers then management will become very difficult (create containers / stop containers / start containers)

=> To overcome these problems we will use Docker Compose.

=> Docker Compose is used to manage Multi - Container Based applications.

=> In docker compose, using single command we can create / stop / start multiple containers at a time.

==========
What is docker-compose.yml file ?
==========

=> docker-compose.yml file is used to specify containers information.

=> The default file name is docker-compose.yml (we can change it).

=> docker-compose.yml file contains below 4 sections

version : It represents compose yml version

services : It represents containers information (image-name, port-mapping etc..)

networks : Represents docker network to run our containers

volumes : Represents containers storage location

==========
Docker Compose Setup
==========

```
# install docker compose

sudo curl -L "https://github.com/docker/compose/releases/download/1.24.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose

# Check docker compose is installed or not

$ docker-compose --version
```

==========
Spring Boot with MySQL DB using Docker-Compose
==========

```
version: "3"

services:

        application:
          image: spring-boot-mysql-app
          ports:
```

```
           - 8080:8080
          networks:
           - springboot-db-net
          depends_on:
           - mysqldb
          volumes:
           - /data/springboot-app

        mysqldb:
          image: mysql:8
          networks:
           - springboot-db-net
          environment:
           - MYSQL_ROOT_PASSWORD=root
           - MYSQL_DATABASE=sbms
          volumes:
           - /data/mysql

networks:
 springboot-db-net


================================
 Application Execution Process
================================

# clone git repo
$ git clone https://github.com/ashokitschool/spring-boot-mysql-docker-compose.git

# go inside project directory
$ cd spring-boot-mysql-docker-compose

# build project using maven
$ mvn clean package -DskipTests=true

# build docker image
$ docker build -t spring-boot-mysql-app .

# check docker images
$ docker images

# create docker containers using docker-compose
$ docker-compose up -d

# check containers created
$ docker-compose ps

========================================
Open postman to test the api endpoints
========================================

 POST :: http://public-ip:8080/add

 add the below sample JSON in request body.

  {
    â€œbookIdâ€: 10101,
    â€œbookNameâ€: â€œJavaâ€,
    â€œauthorâ€: â€œGoslingâ€
  }

 GET :: http://public-ip:8080/fetch


# stop docker containers running
```

```
$ docker-compose stop

# start docker containers running
$ docker-compose start

# delete docker containers using docker-compose
$ docker-compose down
```

==========
Docker summary
==========

1) Challenges in Application Execution Process

2) What is Containerization

3) Life Without Docker vs Life with Docker

4) What is Docker

5) Docker Architecture

6) Dockerfile

7) Docker Image

8) Docker Registry

9) Docker Container

10) Docker Commands

11) Dockerfile KEYWORDS

12) Writing Dockerfile

13) Creating Docker Images

14) Creating Docker Containers

15) Pushing images to Docker Hub

16) Docker Compose