DOCKER

It allows us to package different aspect of application at one place called containers.

Docker Image

It is a lightweight, standalone, and executable software package that includes everything needed to run an application, such as code, runtime, libraries, and dependencies.

Docker Container

It is an instance of a Docker image. It runs the application in an isolated environment, ensuring that it has its own file system, networking, and process space.

How Docker Image and Docker Container are related?

The relationship between Docker images and containers can be likened to the relationship between classes and objects in object-oriented programming. A Docker image is like a class, defining the structure and behaviour, while a Docker container is like an object, representing a specific instance of that image, running in isolation.

Docker File

It is a simple text file that contains instructions on how to build a Docker image.

Docker Registry

It is a storage and sharing platform for Docker images.

How does Docker work?

1. Building an Image: Developers create a Dockerfile, defining the setup and dependencies required for their application. Then they use the Docker CLI (Command Line Interface) to build the Docker image from the Dockerfile.
2. Distributing Images: Docker images can be stored in a centralized registry, like Docker Hub, or private repositories. From there, other developers or systems can easily pull the images to their local environment.
3. Running Containers: Once an image is available on the local system, developers can use the Docker CLI to run containers based on that image. Each container is an isolated environment where the application can execute.

Docker Commands. (Actual commands are not mentioned)

1. Docker pull : pulls an image from docker repository. Tag is just like version. By default latest version is pulled.
2. Docker images: lists only the Docker images that are currently available and actively being used by running or paused containers.
3. Docker images –a: lists all Docker images, including those that are not in use by any container.
4. Docker run : used to create and run a new Docker container from a Docker image.
5. Docker ps : used to lists only the currently running containers on your local machine.
6. Docker ps –a: lists all containers, including both running and stopped containers.
7. Docker stop : used to stop one or more running Docker containers.
8. Docker rm: used to remove one or more Docker containers from your local machine.
9. Docker inspect: used to retrieve detailed information about a Docker object, such as a container, image, network, volume, or other Docker entities.
10. Docker logs: used to view the logs generated by a running Docker container.
11. Docker exec: It allows you to access and interact with a container's shell or execute any other command within the container's environment.
12. Exit: used to exit from an interactive shell session or terminal.
13. Docker stop: used to gracefully stop a running Docker container.
14. Docker rmi: used to remove one or more Docker images from your local machine.
15. Docker prune: removes unused resources from your Docker environment but does not remove images that are being used by any container.
16. Docker prune –a : removes all unused resources, including images that are not being used by any container (dangling images) and images that are not being used by any other images (dangling image layers).
17. Docker run: used to create and start a new Docker container based on a specified Docker image.
18. Docker build: used to create a Docker image from a specified Dockerfile.
19. Docker push: used to upload Docker images from your local machine to a container registry, making them available for others to download and use.

Steps for building a docker image.

1. Create a Dockerfile: The Dockerfile is a text file that contains a set of instructions for building the image. It defines the base image, sets up the environment, copies application code, and configures the image.
2. Choose a Base Image: The Dockerfile starts with selecting a base image that serves as the foundation for your image. Base images are typically minimal operating systems or specific language runtimes (e.g., Python, Node.js).
3. Configure the Environment: The Dockerfile can specify environment variables, set the working directory, and define other configuration settings needed for your application to run correctly.
4. Copy Application Code: If your application requires source code or files, use the Dockerfile to copy them into the image from your local machine or a remote repository.
5. Install Dependencies: If your application relies on specific libraries or packages, use the Dockerfile to install them in the image.
6. Run Build Commands: The Dockerfile can include commands to run build processes or compile code necessary for your application.
7. Expose Ports (Optional): If your application listens on specific ports, use the Dockerfile to expose those ports in the image, so they can be accessed when running a container.
8. Cleanup (Optional): In the Dockerfile, you can clean up unnecessary files or artifacts to reduce the size of the final image.
9. Build the Image: Once you have created the Dockerfile, you use the docker build command to build the Docker image. This command reads the instructions from the Dockerfile and generates the image based on those instructions.

Example:

# Use the official OpenJDK 11 image as the base image  
FROM openjdk:11

# Set the working directory inside the container  
WORKDIR /app

# Copy the packaged Spring Boot JAR file into the container  
COPY target/my-spring-app.jar /app/app.jar

# Expose the port that the Spring Boot application listens on (assuming it's 8080)  
EXPOSE 8080

# Set the command to run the Spring Boot application when the container starts  
CMD ["java", "-jar", "app.jar"]

Explanation:

The FROM instruction specifies the base image to use. In this case, we use the official OpenJDK 11 image from Docker Hub, which contains the Java Runtime Environment needed to run the Spring Boot application.

The WORKDIR instruction sets the working directory inside the container. In this example, we set it to /app, where we will copy the application files.

The COPY instruction copies the packaged Spring Boot JAR file (assuming it's named my-spring-app.jar) from the target directory of your local project into the container's /app directory.

The EXPOSE instruction exposes port 8080, which is the default port that Spring Boot applications often use to listen for incoming requests.

The CMD instruction specifies the command to run the Spring Boot application when the container starts. In this case, we use the java -jar app.jar command to execute the JAR file we copied earlier.

Building the Image:

To build the Docker image from this Dockerfile, navigate to the directory containing the Dockerfile and your Spring Boot JAR file, and run the following command:

docker build -t my-spring-app-image

Replace my-spring-app-image with the desired name and tag for your Docker image.

Running the Container:

After building the image, you can run the container based on the image using the docker run command:

docker run -p 8080:8080 my-spring-app-image

Note:

In a Dockerfile, ENTRYPOINT is an instruction that specifies the command to be executed when a container is started from the image. It defines the default application or process that should run when the container is launched.

DOCKER COMPOSE

While working with multiple containers we may have to initialise all containers or run docker command multiple times or we want to establish relationship b/w containers then this has to be managed via docker commands, but sometimes it is not an easy task for us to run those commands manually.

Here docker compose is a tool which is used for running multiple Containers as single service.

Here each container runs in isolation but can interact with each other.

All docker compose files are YAML files.

Note:-

Docker Container takes care of creating a common network.

This statement is useful when we want to run multiple Containers in a single in a single network so that these Containers can interact with each other.

Alternative tedious way is to specify network in docker run command.

Example:

version: '3.8'

services:

backend:

build:

context: ./backend

dockerfile: Dockerfile

ports:

- "8080:8080"

depends\_on:

- mysql

mysql:

image: mysql:5.7

environment:

MYSQL\_ROOT\_PASSWORD: mypassword

MYSQL\_DATABASE: mydb

MYSQL\_USER: myuser

MYSQL\_PASSWORD: mypassword

volumes:

- mysql-data:/var/lib/mysql

volumes:

mysql-data:

Explanation

In the docker-compose.yml file, we define two services: backend and mysql.

The backend service:

It builds the Docker image for the Spring Boot application using the Dockerfile from the backend directory. Maps port 8080 from the container to port 8080 on the host, allowing access to the Spring Boot application. ‘depends\_on’ specifies that the backend service depends on the mysql service, ensuring that the database is running before the application starts.

The mysql service:

It uses the official MySQL 5.7 image from Docker Hub. Defines environment variables for configuring the MySQL database (root password, database name, user, and password). Mounts a named volume mysql-data to persist the MySQL data between container restarts.

To build and run the entire application stack, run the following command in the project directory:

docker-compose up

Docker Compose will create two containers, one for the Spring Boot backend and one for the MySQL database. The containers will be connected in the same Docker network, allowing them to communicate. The Spring Boot application will be accessible at <http://localhost:8080>.

Docker Compose Health Checks

Docker Maven JIB Plugin