Docker

Docker is a platform for developing, shipping, and running applications in containers. It provides an efficient way to package applications with all their dependencies into a standardized unit for deployment. Here's an overview of key concepts and components in Docker:

- Containers: Containers are lightweight and portable units that encapsulate software and its dependencies. They provide isolation and consistency across different environments, making it easier to deploy applications reliably.

- Images: Docker images are read-only templates used to create containers. They contain everything needed to run an application, including the code, runtime, libraries, and dependencies. Images are built from Dockerfiles, which are text files that define the steps to create the image.

- Docker Engine: Docker Engine is the core component of Docker. It is a client-server application that runs on the host machine and manages containers and images. The Docker Engine consists of a daemon process (dockerd) that runs in the background and a command-line interface (CLI) client (docker) used to interact with the daemon.

- Dockerfile: A Dockerfile is a text file that contains instructions for building a Docker image. It specifies the base image, commands to install dependencies, configure the environment, and define how to run the application. Dockerfiles follow a simple syntax and can be version-controlled alongside application code.

- Registry: Docker Registry is a storage and distribution service for Docker images. It allows users to store and share images publicly or privately. The Docker Hub is the default public registry provided by Docker, while organizations can set up their private registries for internal use.

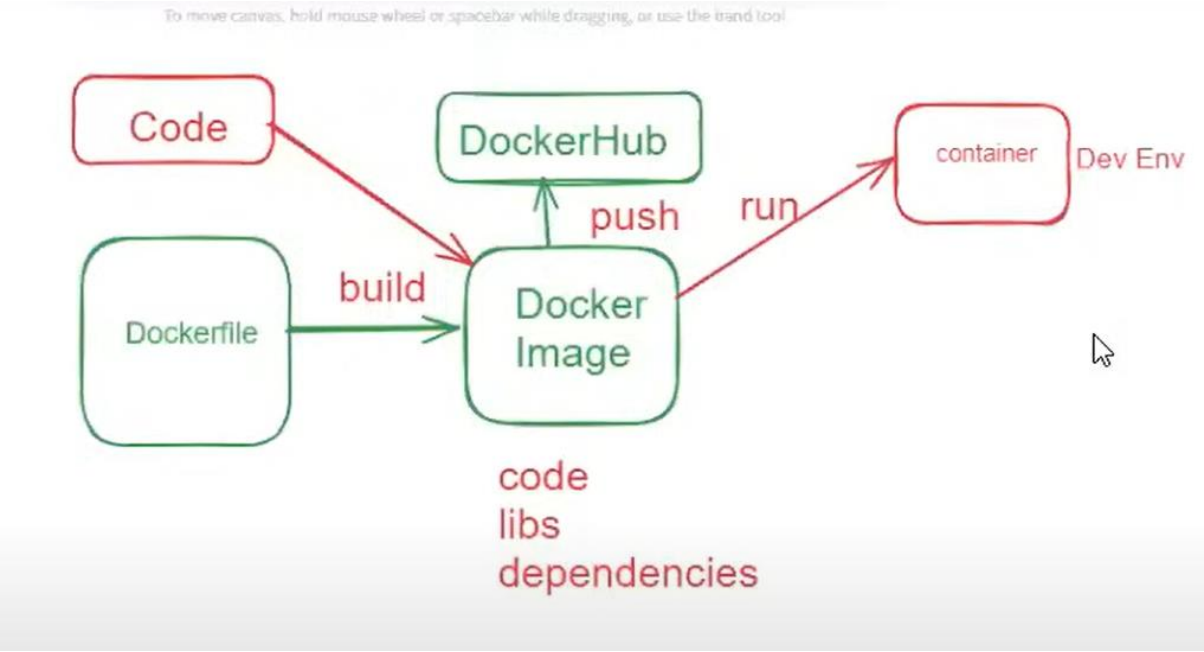
- Docker Compose: Docker Compose is a tool for defining and running multi-container Docker applications. It uses a YAML file (docker-compose.yml) to configure the services, networks, and volumes required for the application. Compose simplifies the orchestration of complex applications by managing multiple containers as a single unit.

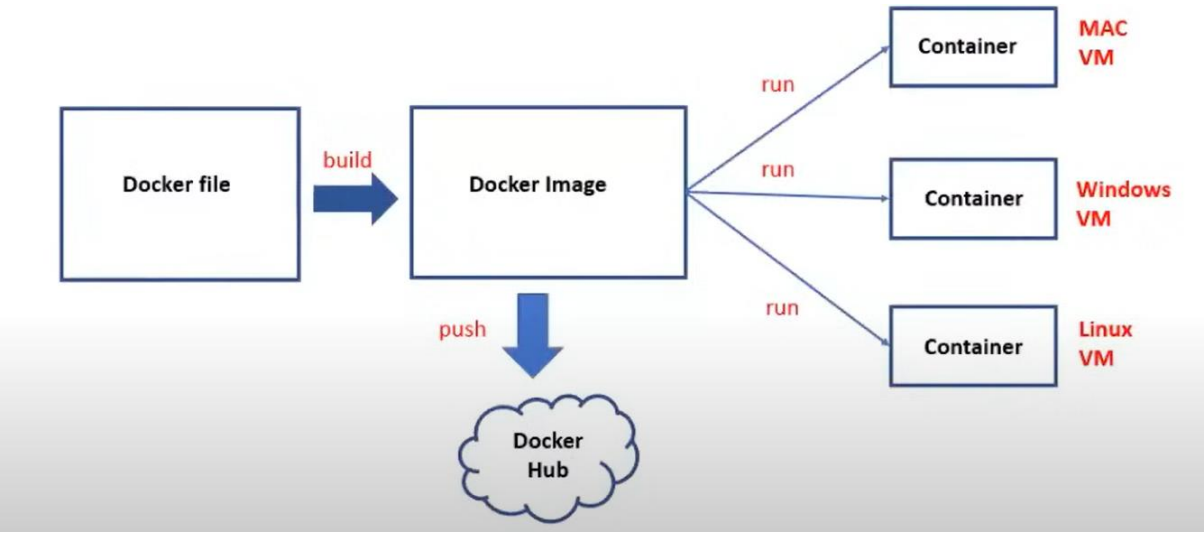
- Docker Swarm: Docker Swarm is a native clustering and orchestration tool for Docker. It enables the creation and management of a cluster of Docker hosts, allowing users to deploy and scale applications across multiple machines. Swarm provides features such as service discovery, load balancing, and rolling updates.

- Docker Desktop: Docker Desktop is an application for Windows and macOS that provides an easy-to-use interface for developing and deploying Docker applications. It includes Docker Engine, Docker CLI, Docker Compose, and other tools necessary for building and running containers on a local development machine.

Docker Architecture

Along with the code, it is the developer's responsibility to write the docker file. Docker file is the file where it contains all the configuration to build the docker image. Docker image contains all the code, libraries, dependencies. The docker image is pushed to the docker hub. And when the image is run,it will get converted to container.





Basic Docker Commands

- docker run: Create and start a container from an image.

  docker run <image_name>

- docker ps: List running containers.

  docker ps

- docker ps -a: List all containers (including stopped ones).

  docker ps -a

- docker stop: Stop a running container.

  docker stop <container_id or container_name>

- docker start: Start a stopped container.

  docker start <container_id or container_name>

- docker rm: Remove one or more containers.

  docker rm <container_id or container_name>

- docker images: List available images.

  docker images

- docker rmi: Remove one or more images.

  docker rmi <image_id or image_name>

- docker pull: Pull an image or a repository from a registry.

  docker pull <image_name>

- docker exec: Run a command in a running container.

  docker exec <container_id or container_name> <command>

- docker logs: Fetch the logs of a container.

  docker logs <container_id or container_name>

- docker build: Build an image from a Dockerfile.

  docker build -t <image_name> <path_to_dockerfile>

- docker system prune -a: Delete the Images and Containers which are no longer used.
- docker build -t app2 -f dockerfile2 . - If u want to build the image by using the different dockerfile name.

- docker-compose up – Create the container.
- docker-compose down – Delete the crested container.
- docker-compose up -d – Create the container in detached mode.

Docker Keywords

- **FROM** – It is used to specify base image required for our application Eg : FROM openjdk:17
- **MAINTAINER** – It is used to specify author of Dockerfile Eg:MAINTAINER <suraj >
- **COPY** – It is used to copy the files from host machine to container machine.

  Eg: COPY <SRC> <DEST>

  COPY target/app.jar usr/app/tomcat/webapp.war

- **RUN** -It is used to create instructions while running docker image.

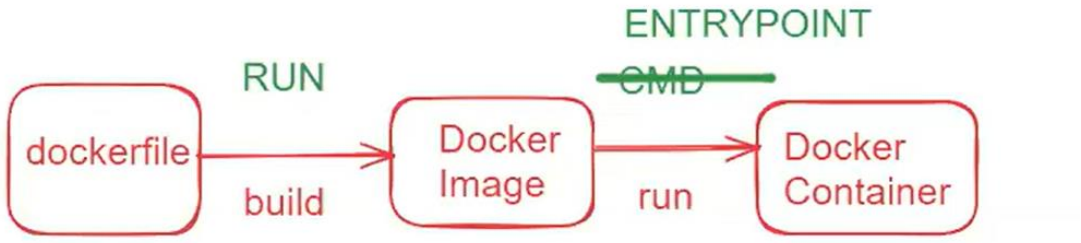
  RUN 'sudo yum install git'

We can give multiple run instructions in a single Docker image. It is used while creating images. If we have multiple RUN commands, all RUN commands will be executed from top to bottom.

- **CMD** - It is used to execute instructions while creating docker container

  CMD "java -jar <jar-file>"

If we have multiple CMD, then last CMD will be executed ie The CMD instruction will be overrided. ENTRYPOINT is used as alternative for CMD



- **EXPOSE** – It is used to specify the container port number but not to change the container port number. Used for documentation purposes only. If the container port number is 8081 and the "EXPOSE 1111" is written. The container port will be 8081 only but to specify to other people, EXPOSE is used.
- **WORKDIR** – It is used to specify the working directory (path change)

Demo Project was done to illustrate the containerization of application.

Dependencies used for the project.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-mongodb</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
<groupId>org.springdoc</groupId>
<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
<version>2.2.0</version>
</dependency>

<dependency>
<groupId>org.sonarsource.scanner.maven</groupId>
<artifactId>sonar-maven-plugin</artifactId>
<version>3.10.0.2594</version>
</dependency>

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
<groupId>io.micrometer</groupId>
<artifactId>micrometer-registry-prometheus</artifactId>
<scope>runtime</scope></dependency>
```

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter</artifactId>
</dependency>

<dependency>
<groupId>org.springdoc</groupId>
<artifactId>springdoc-openapi-ui</artifactId>
<version>1.6.9</version>
</dependency>

<dependency>
<groupId>io.springfox</groupId>
<artifactId>springfox-boot-starter</artifactId>
<version>3.0.0</version>
</dependency>

<dependency>
<groupId>com.fasterxml</groupId>
<artifactId>classmate</artifactId>
<version>1.5.1</version> <!-- Use the latest version -->
</dependency>

<dependency>
<groupId>com.github.ulisesbocchio</groupId>
<artifactId>jasypt-spring-boot-starter</artifactId>
<version>3.0.5</version>
</dependency>
```

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.9.2</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.9.2</version>

</dependency>

Application properties file for giving the credentials for MongoDB database.

Steps for connecting to MongoDB database:

- Create the MongoDB Atlas account.
- Create the username and password.
- Inside Atlas, add this IP Address (0.0.0.0/0 (includes your current IP address)) into Security > Network Address. In theory this should allow me to connect to the database from any IP address.
- Then created a collection called "employeedb".
- If we click on my cluster and then on the connect button, it asks with which modality I want to connect. I choose "Connect your application", and then we must select the Driver and the Version. I choose respectively "Java" and "3.6 or later" (I'm not sure if it's the correct version, the alternatives are 3.4 or 3.3). And finally, it shows me the connection string which is:
- mongodb+srv://admin:<password>@umadit-obxpb.mongodb.net/test?retryWrites=true&w=majority
- Take the URI string and paste in the application properties file.
- Paste the password and database name in the string.
- Make the post operation initially to display the database in the MongoDB Atlas account.

spring.data.mongodb.uri=mongodb+srv://suraj:35AKQ3oXehQhgITb@cluster0.da5t9wf.mongodb.net/employeedb?retryWrites=true&w=majority&appName=Cluster0

Dockerfile

This Dockerfile is used to build a Docker image for a Spring Boot application.

- FROM openjdk:17: This line specifies the base image for this Docker image. In this case, it's using the OpenJDK 17 image as the base. This means that the image will contain the Java runtime environment needed to run the Spring Boot application.

- ADD target/springboot-docker-compose.jar springboot-docker-compose.jar: This line adds the Spring Boot application JAR file (springboot-docker-compose.jar) from the target directory of your project to the root directory of the Docker image. This assumes that you have already built the Spring Boot application using Maven or Gradle and the resulting JAR file is located in the target directory.

- ENTRYPOINT ["java","-jar","springboot-docker-compose.jar"]: This line specifies the command that will be executed when a container is started from this Docker image. It runs the Java executable (java) with the -jar option, specifying the Spring Boot application JAR file (springboot-docker-compose.jar) as the argument. This means that when the container starts, it will automatically run the Spring Boot application.

```
FROM openjdk:17

ADD target/springboot-docker-compose.jar springboot-docker-compose.jar

ENTRYPOINT ["java","-jar","springboot-docker-compose.jar"]
```

Final Name in pom.xml file

Whenever the application is built, the jar file will be built with the name given in the pom.xml before the build

```
<finalName>springboot-docker-compose</finalName>

</build>
```

Docker commands to run the application in the container.

- Mvn clean package - To build the jar file of the application.
- docker build -t spring-docker-compose -This command tells Docker to build a Docker image using the Dockerfile in the current directory (.) and tag it with the name spring-docker-compose using the -t flag. Make sure you're in the directory where your Dockerfile is located when running this command.
- docker run -d -p 8089:8089 <image id>
- The container will be create and run

- Verify it by docker ps command

Hit the API with 8089 port for the desired result

Running microservices in Docker Container

Demo Project was done to illustrate the containerization of the services.

Dependencies used for the all the services.

<dependencies>

<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>

</dependencies>

<dependencyManagement>
<dependencies>

<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>${spring-cloud.version}</version>
<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

Dockerfile of all the services

User-Service:

FROM openjdk:17

ADD target/user-service.jar user-service.jar

ENTRYPOINT ["java","-jar","user-service.jar"]

Contact-Service:

FROM openjdk:17

ADD target/contact-service.jar contact-service.jar

ENTRYPOINT ["java","-jar","contact-service.jar"]

Eureka -server:

FROM openjdk:17

ADD target/eureka-server.jar eureka-server.jar

ENTRYPOINT ["java","-jar","eureka-server.jar"]

Api-Gateway:

FROM openjdk:17

ADD target/contact-service.jar contact-service.jar

ENTRYPOINT ["java","-jar","contact-service.jar"]

Configuration of name in pom file

User-Service:

<finalName>user-service</finalName>

</build>

Contact-Service:

<finalName>contact-service</finalName>

</build>

Api-gateway:

<finalName>api-gateway</finalName>

</build>

Eureka-server:

<finalName>eureka-server</finalName>

</build>

Build the images with these commands in each service.

mvn clean package

docker build -t <service-name> .

docker run -d -p internalport:containerport <image-id>

Docker-Compose file

Whenever there are multiple containers to be created, Use docker-compose file

```
version: "3.8"
services:
  user-service:
    image: user-service:latest
    container_name: "user-service"
    ports:
      - "9001:9001"

  contact-service:
```

```
  image: contact-service:latest
  container_name: "contact-service"
  ports:
    - "9002:9002"

api-gateway:
  image: api-gateway:latest
  container_name: "api-gateway"
  ports:
    - "8999:8999"

eureka-server:
  image: eureka-server:latest
  container_name: "eureka-server"
  ports:
    - "8761:8761"
```

Go to the directory of the docker-compose file and run “docker-compose up” file. All the services will be started by the Docker.

Hit the Apis of the services and the desired result will be obtained.