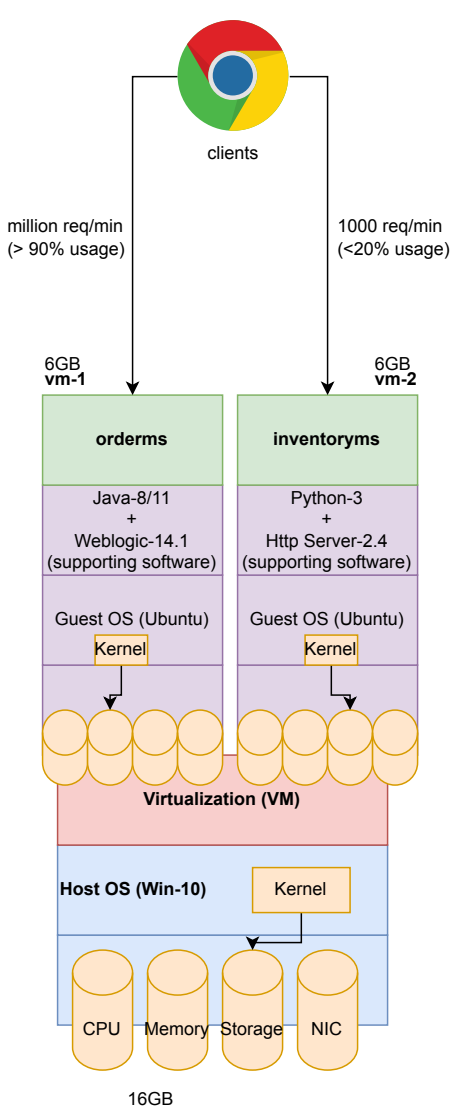
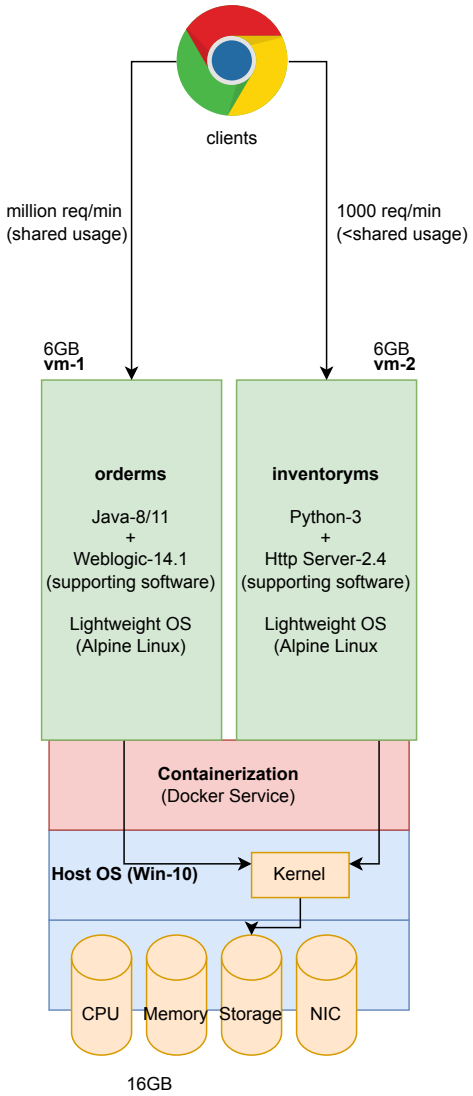
VM vs Docker

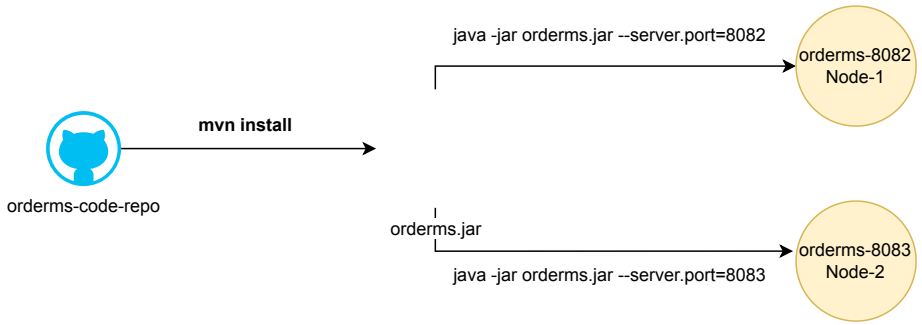
Issues with VM:

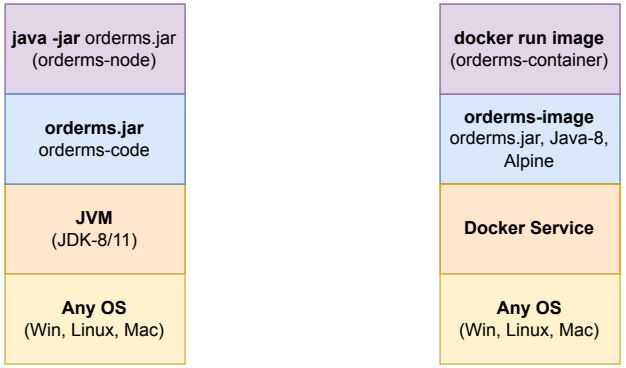
1. Resource sharing not possible
2. Usage of resource is not optimal
3. Software version mismatch
4. Longer startup time



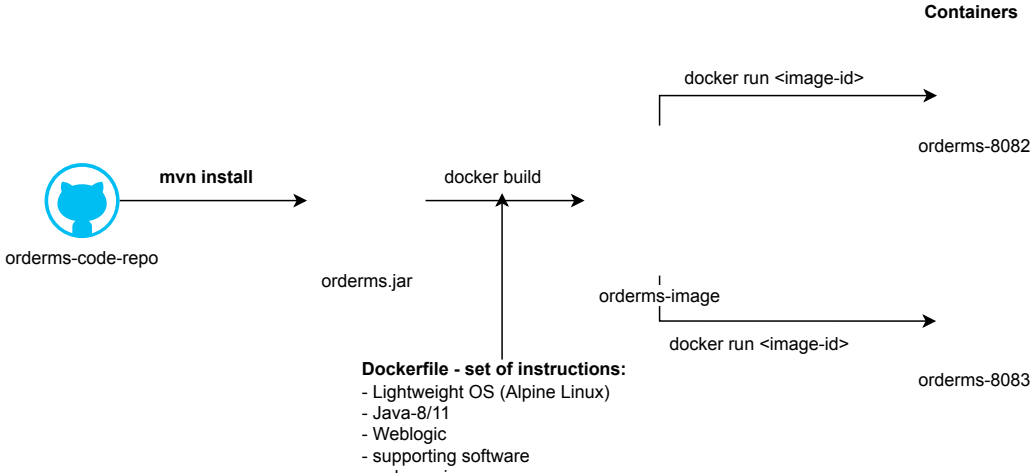


VM:


java -jar orderms.jar --server.port=8082
orderms-8082
Node-1
mvn install
orderms-code-repo
orderms.jar
orderms-8083
Node-2
java -jar orderms.jar --server.port=8083


java -jar orderms.jar
(orderms-node)
orderms.jar
orderms-code
JVM
(JDK-8/11)
Any OS
(Win, Linux, Mac)

docker run image
(orderms-container)
orderms-image
orderms.jar, Java-8,
Alpine
Docker Service
Any OS
(Win, Linux, Mac)


Containers
docker run <image-id>
orderms-8082
mvn install
docker build
orderms-code-repo
orderms.jar
orderms-image
docker run <image-id>
orderms-8083
Dockerfile - set of instructions:
- Lightweight OS (Alpine Linux)
- Java-8/11
- Weblogic
- supporting software

- orderms.jar
- java -jar orderms.jar

Dockerfile

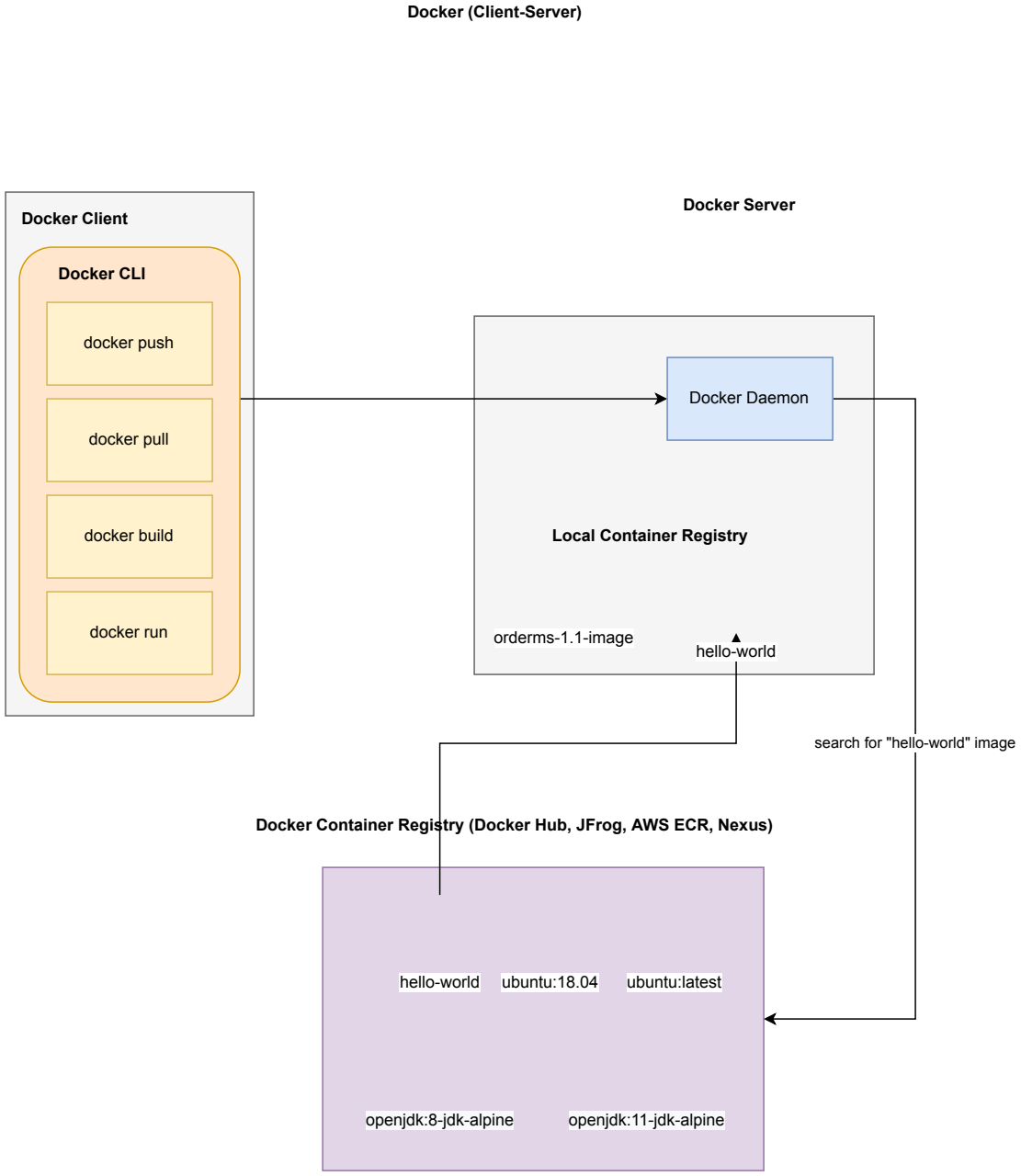
```
# With default name of "Dockerfile"
docker image build -t misbaharchitect/firstrepo:orderms-1.1 .

# With custom name of Dockerfile as "MyDockerfile"
docker image build -t misbaharchitect/firstrepo:orderms-1.1 -f MyDockerfile
```

Dockerfile:

```
FROM baseImage
RUN mkdir /app
COPY orderms.jar orderms.jar
ENTRYPOINT ["java", "-jar", "orderms.jar"]
```

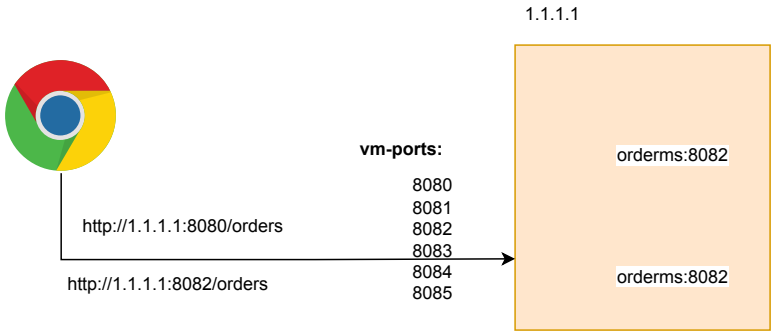
```
FROM ubuntu:latest
RUN sudo apt install openjdk-8-jre-headless
COPY orderms.jar orderms.jar
ENTRYPOINT ["java", "-jar", "orderms.jar"]
```

```
FROM openjdk:8-jdk-alpine
COPY orderms.jar orderms.jar
ENTRYPOINT ["java", "-jar", "orderms.jar"]
```


Docker (Client-Server)
Docker Client
Docker CLI
docker push
docker pull
docker build
docker run
Docker Server
Docker Daemon
Local Container Registry
orderms-1.1-image
hello-world
search for "hello-world" image
Docker Container Registry (Docker Hub, JFrog, AWS ECR, Nexus)
hello-world
ubuntu:18.04
ubuntu:latest
openjdk:8-jdk-alpine
openjdk:11-jdk-alpine

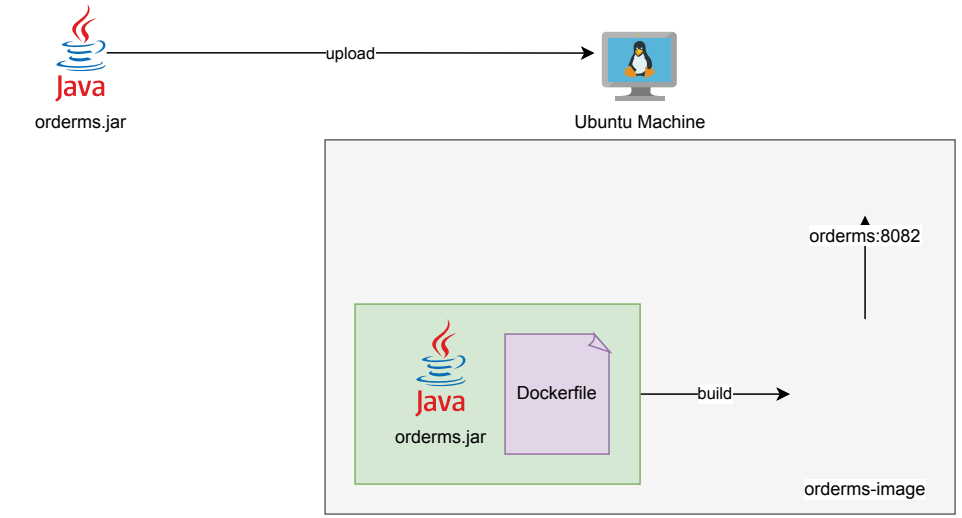
Day-2

Port Forwarding




```
docker run -p <vm-port>:<docker-port> <orderms-image>
docker run -p 8080:8082 <orderms-image>
docker run -p 8082:8082 <orderms-image>
```

Built on local machine



Day-3

Multistage Build

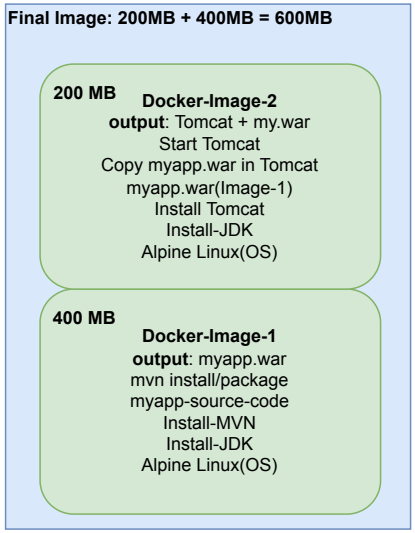
External Tomcat

$tomcat_home/webapp/myapp.war

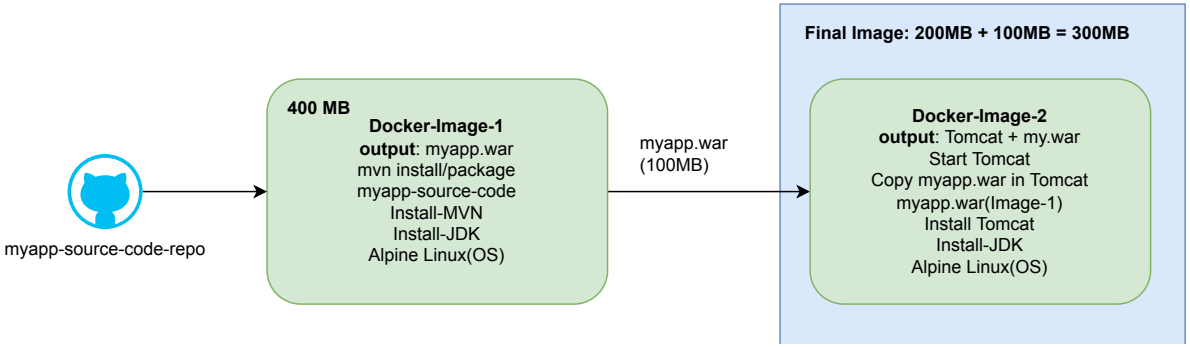
Normal Build

Ideally, 200 MB + myapp.war (100 MB) = 300 MB

Huge Docker Image



Multistage Build



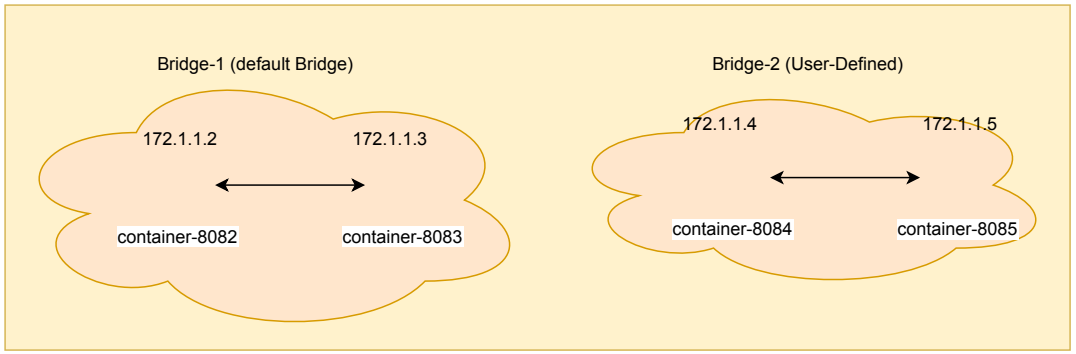
Docker Network

1. Bridge(default)
2. Host
3. Overlay
4. None (Custom)

1. Bridge

(Single-House Networking): It's the default Network.
docker run -p 8081:8082 my-image
curl localhost:8081/hello

ubuntu-machine(host)



log in to container-8082:
ping 172.1.1.3 // works
ping container-8083 // does not work

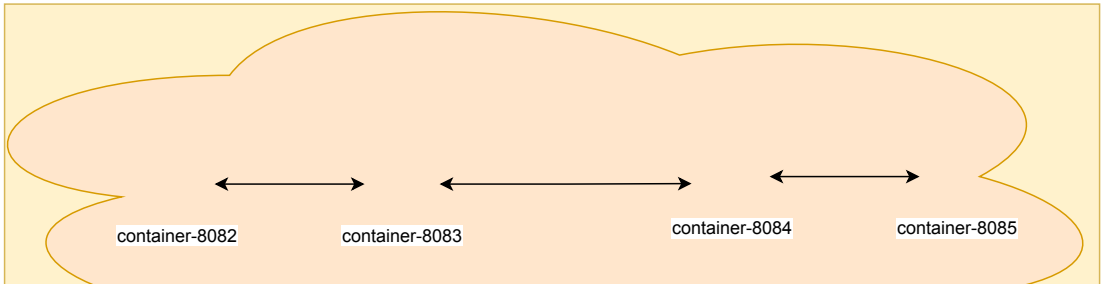
container-8083 and container-8084 can not talk to each other

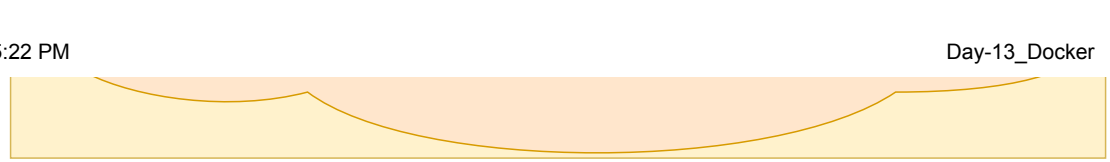
log in to container-8084:
ping 172.1.1.5 // works
ping container-8085 // works

2. Host

(No Port mapping required)
It uses hot machine network/port.
docker run --network host my-image
curl localhost:8080/hello

ubuntu-machine(host)

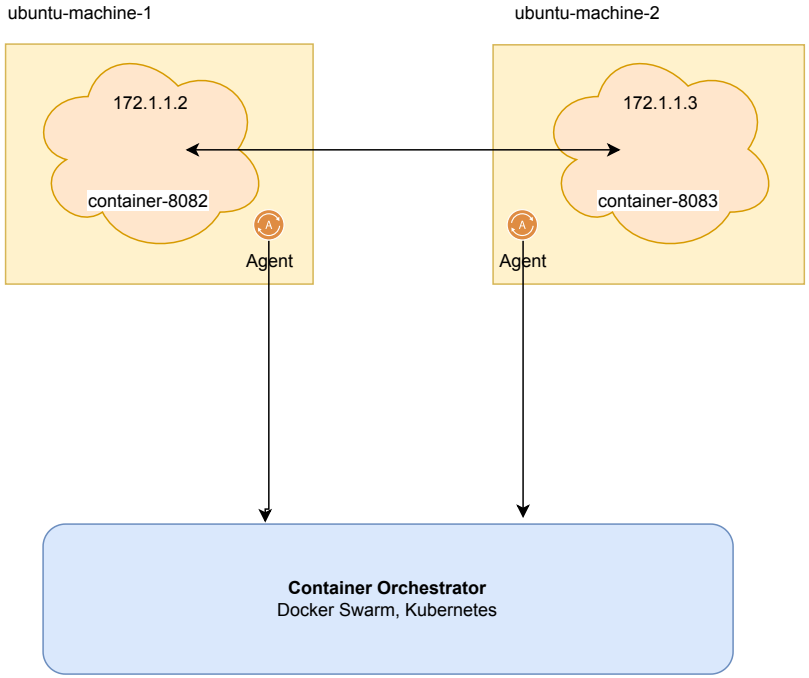




Docker Network

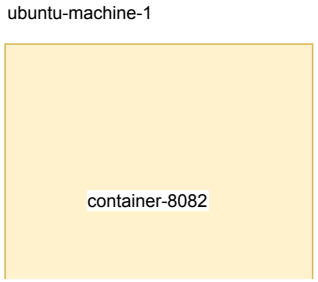
3. Overlay

It's distributed network among multiple Docker hosts.
docker run -p 8081:8082 --network my-overlay-network my-image
curl localhost:8081/hello



4. None

It disables the Docker Network
docker run --network none my-image



Day-4

Docker Compose

1. Build Docker image postgrems(microservice)
2. Get postgres DB image
3. New Network (custom)
4. Run postgres container
5. Attach postgres container to new n/w
6. Run postgresms container
7. Attach postgresms container to the new network
8. Integrate postgresms with postgres

ubuntu machine: