

Content List

[Content List 1](#_Toc167847280)

[Monolithic vs Microservices Architecture 1](#_Toc167847281)

[Docker Networks 5](#_Toc167847282)

[Some Docker Network Commands 6](#_Toc167847283)

[The none/null network in Docker 11](#_Toc167847284)

[Docker 5 12](#_Toc167847285)

[Docker Volumes 12](#_Toc167847286)

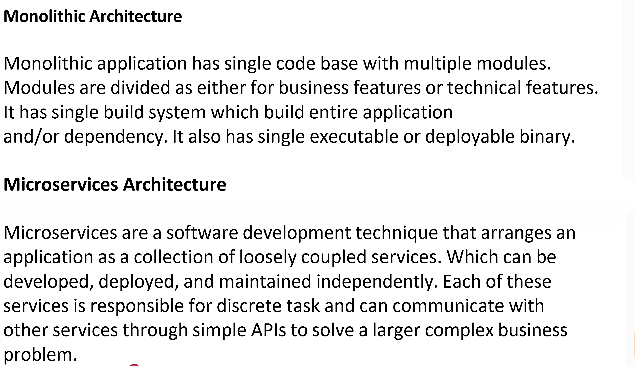
[What are Docker Volumes? 13](#_Toc167847287)

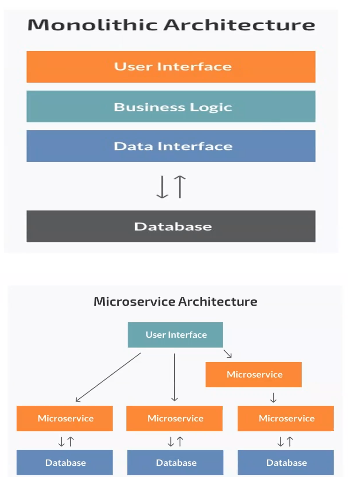
[Docker Persistent Volumes (Docker Named Volumes) 17](#_Toc167847288)

[External Volumes 18](#_Toc167847289)

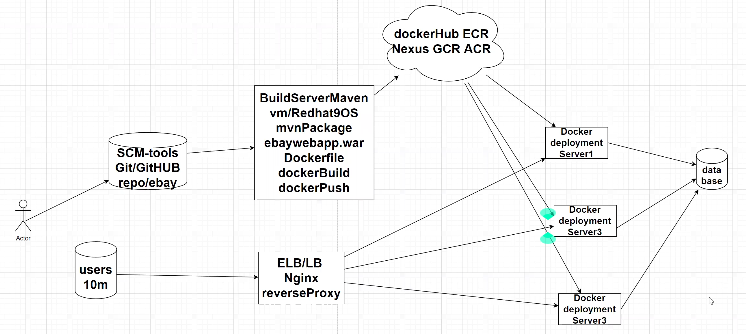
[Docker Compose 20](#_Toc167847290)

Monolithic vs Microservices Architecture



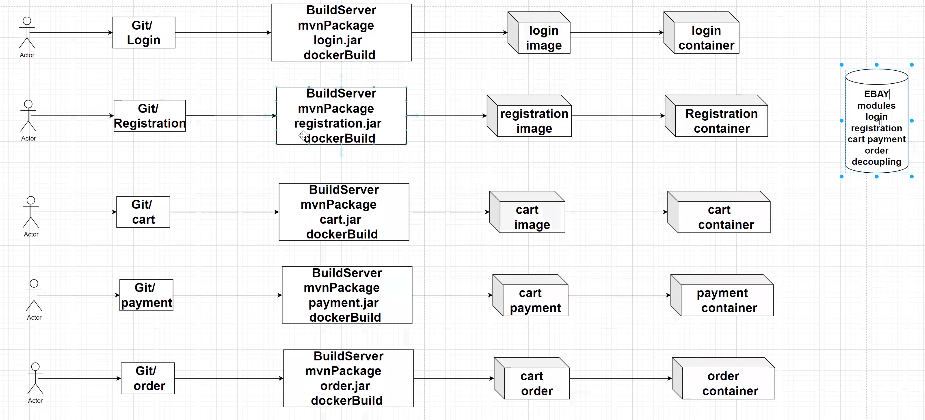


**Monolithic Architecture - eBay example**

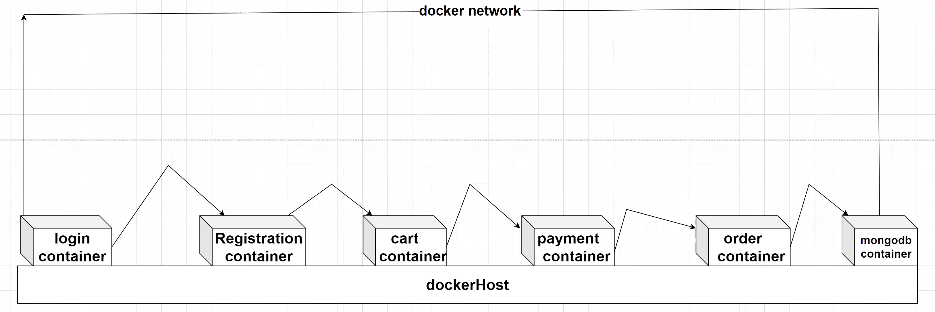


Above illustrates the deployment of an application using the monolithic architecture. The whole software is deployed as one entire module.

Below depicts the decoupling of the same application into microservices, this permits easy development, deployment and maintenance.



These containers can therefore be deployed in a dockerHost with a docker network for inter-service communication as shown below:



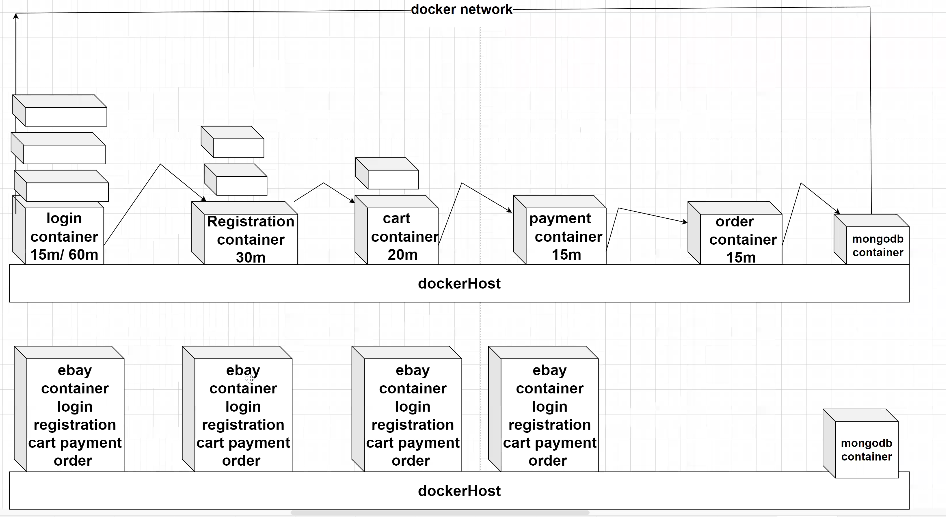
**Monolithic**

* Pros
  + Simple to develop
  + Simple to deploy single package (war/jar/ear, etc...)
  + Easy to debug and error tracing
  + Simple to test
* Cons
* Difficult to understand and modify
* Tightly coupled
* Redeploy entire app on each update
* Single bug can bring down entire application
* Scaling the application is difficult. If we need to scale only few features/modules will end up scaling entire app as its single package.
* Changes on one section(module/features) of the code can cause impact on the other section of the code as it is a single code base.

**Microservices**

* Pros
  + Loosely coupled
  + Easy to understand and modify as it’s small codebase
  + Better deployments as each service (feature/module) can be deployed independently
  + Each service can be scaled independently as each service is a separate code base(Repository)
* Cons
* **Communication between services is complex**: Because microservices are independent, communication between them needs to be carefully handled using mechanisms like REST APIs.
* **Integration testing is difficult**: Testing a microservices application can be cumbersome because each service needs to be deployed before integration testing can occur. In a monolithic application, you would just need to launch the entire application on a server with the underlying database. With microservices, each dependent service needs to be deployed before testing can occur.
* **Debugging problems can be harder:** Debugging microservices applications can be more difficult because each service has its own set of logs and code to analyse.
* **Deployment challenges**: The application may need coordination

In the scenario below, individual microservices are scaled as need arises, depending on the number of users accessing the various modules at a time, compared to the second case where the whole application has to be scaled to accommodate the increase in access requests.



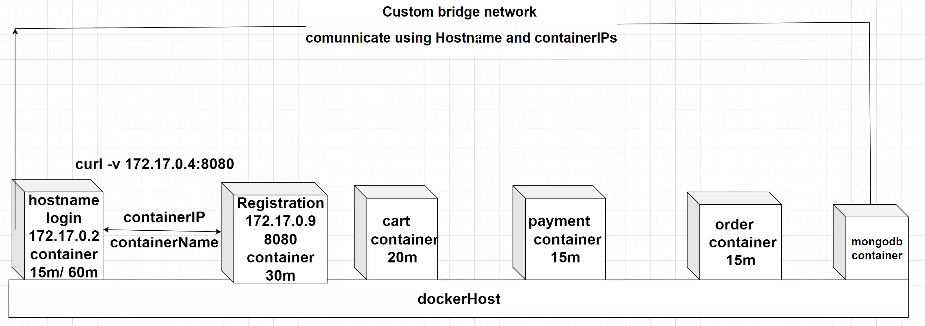
Docker Networks

✅ Containers communicate using IP addresses and or container names on an established network.

**Definition**:

Docker Networks refer to virtual networks created within the Docker ecosystem, enabling communication between Docker containers and other network resources, facilitating efficient and isolated data exchange among various components of distributed applications.

✅With default bridge network, containers can only communicate using IP addresses, meanwhile with a Custom bridge networks, docker containers can communicate using both IPs and Hostnames.



✨ Custom bridge network is the better option because it permits the containers to connect via IP and Hostname, in case containers are destroyed and recreated, the IP addresses change meanwhile the Hostnames will remain the same. Hence the network will not be interrupted.

Some Docker Network Commands

docker network ls #list the present networks

docker network create tesla # creating a custom network for tesla

docker network create -d bridge ebay # create a bridge network for eBay

docker network ls # see the created networks

👉Docker supports; Default bridge, Custom Bridge, Host and None networks

docker inspect app # Note the ip address

docker exec -it app bash # enter interactive mode using bash

curl -v 172.17.0.4:8080/java-web-app # trying to connect to application using ip or hostname

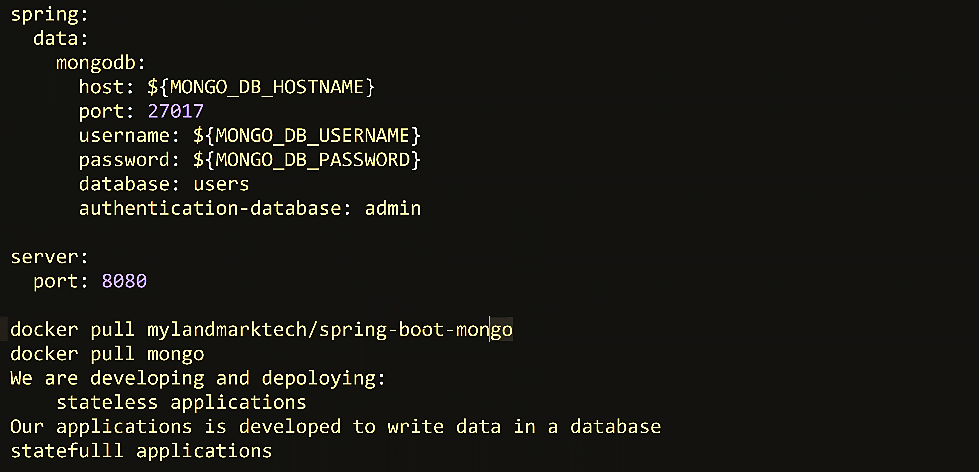
exit # to exit from interactive mode

docker inspect app # to inspect the network for app

**We Support Java based applications:**

**We also support nodeJS and .net based applications**:

Example below is done using the spring-boot-mongo application by our developers.(search for the image in DockerHub). Always read the instructions on how to use an image in dockerhub.



Let us pull the spring-boot-mongo and mongo images from dockerhub

docker rmi -f $( docker images -q ) # force delete all available images

docker images # verify all deleted

docker pull fewaitconsulting/spring-boot-mongo

docker pull mongo # pull the mongodb image

docker images # see the images

#now to run and create containers from the images we can run

docker run --name springapp -d -p 4000:8080 --network tesla -e MONGO\_DB\_USERNAME=devdb -e MONGO\_DB\_PASSWORD='dev@123' -e MONGO\_DB\_HOSTNAME=mongodb fewaitconsulting/spring-boot-mongo

✨ let's break down each part of the docker run command:

1. **docker run**: This is the command used to run a Docker container. It tells Docker to create and start a new container based on the provided image.
2. **--name springapp**: This part of the command specifies the name you want to assign to the container. In this case, the container will be named "springapp".
3. **-d**: This option runs the container in detached mode, meaning it runs in the background and doesn't hold up your terminal session.
4. **-p 4000:8080**: This option publishes a container's port(s) to the host. In this case, it maps port 8080 from the container to port 4000 on the host. So any traffic sent to port 4000 on the host will be forwarded to port 8080 in the container.
5. **--network tesla**: This option specifies the network to which the container should be connected. Docker networks allow containers to communicate with each other.
6. **-e MONGO\_DB\_USERNAME devdb**: This option sets an environment variable in the container. In this case, it sets the **MONGO\_DB\_USERNAME** variable to **devdb**. This could be used by the containerized application to configure its connection to a MongoDB database.
7. **-e MONGO\_DB\_PASSWORD=dev@123**: Similar to the previous option, this sets the **MONGO\_DB\_PASSWORD** environment variable to **dev@123**.
8. **-e MONGO\_DB\_HOSTNAME=mongodb**: Again, this sets an environment variable, **MONGO\_DB\_HOSTNAME**, to **mongodb**. This likely specifies the hostname or IP address of the MongoDB server the containerized application should connect to.
9. **fewaitconsulting/spring-boot-mongo**: This is the name of the Docker image that will be used to create the container. Docker will pull this image from the Docker Hub repository if it's not already available locally. In this case, it appears to be an image named **spring-boot-mongo** provided by the **fewaitconsulting** organization or user on Docker Hub.

So, altogether, this command is telling Docker to create and start a container named "springapp" based on the **fewaitconsulting/spring-boot-mongo** image. It will run in detached mode, publish port 8080 from the container to port 4000 on the host, connect it to the "tesla" network, and set some environment variables related to MongoDB configuration.

curl:-v 44.211.75.85:4000 # with host public ip

Also, get your IP and port number to the web and verify that the application is running. Take note that the database hasn’t been deployed yet, so it won’t work properly.

Now, let us verify how to use the mongo image by checking dockerhub, especially how to pass env variables or credentials like username and passwd.

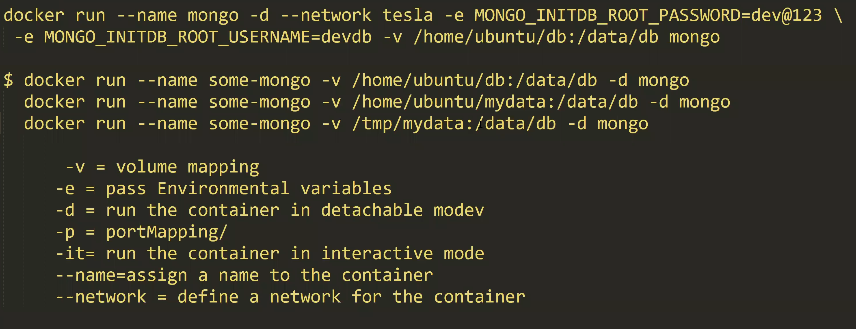
docker run --name mongo -d --network tesla -e MONGO\_ROOT\_PASSWORD=dev@123 -e MONGO\_INITBD\_ROOT\_USERNAME=devdb mongo

docker ps # to verify that the database is running

✨Now, this example has been demonstrated using MongoDB, if in case in your environment you are required to work with any other database, say MySQL, you should follow the same procedure. Get to dockerhub, pull the image, check the notes on how to use the image and proceed as needed.

✨Note that, if one docker container has to talk to another container using ContainerIP/Name(DNS), both have to be under the same docker network.

With databases like our mongo above, we shall be required to specify the directory where to mount our volume. Eg, /data/db ( we can always check image notes for default values and directories).

Some examples below:

**Question**: Can containers running on two separate networks communicate with each other?

**Answer**: No by default

Yes they can be customized to communicate with each other.

**What can we do to enable containers running on different networks to communicate with each other?**

✅ We can use the command “docker exec” to achieve this.

docker exec container1 ping container2 #or we use IPAddress like below

docker exec container1 ping <container2\_IP>

back to our cli for more docker network commands;

docker network ls #always do this to list the available networks

#say we have ebay and tesla networks running

docker inspect ebay #inspect the ebay network and note the subnet IP

docker inspect mongo #inspect mongo and note Gateway and IPAddress

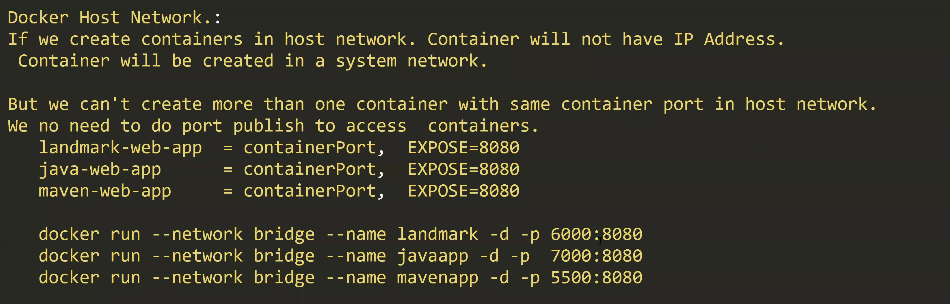
docker inspect springapp #inspectspringapp, note Gateway and IPAddress

docker network –help #lists for you the various network commands

**docker network create/inspect/ls/disconnect/connect/rm/prune**

docker network prune #remove unused networks

docker network rm <networkNameOrID> #removes specified network



But if we deploy containers on host network, we don’t need to specify the port.

Let us pull the hello image from the fewaitconsulting repo on dockerhub

docker pull fewaitconsulting/hello

docker inspect fewaitconsulting/hello #find the ExposedPorts item inside the Config tag



It is exposed on port 80, so if we deploy, it is by default on port 80

#let us verify if port 80 is in use by running

netstat -tulpn # tulpn = tcp, udp, listen port number

#port not in use so we can deploy the hello app without specifying the port number like so;

docker run --network host -d --name hello fewaitconsulting/hello

#docker run --name hello -d -p 90:80 fewaitconsulting/hello # in case port 80 is in use and we want to use port 90 to deploy.

netstat -tulpn #now we can appreciate that port 80 is in use

docker ps # verify running applications

docker inspect hello # note that there is no ip address listed, hostname = docker, cos it assumes host server details

Let us deploy the container now in ebay network

**Note** that we cannot use port 80 again since it is in use, we also have to change the name of container

docker run --network ebay -d -p 90:80 --name hello2 fewaitconsulting/hello

docker inspect hello2 #notice now that that there is an IP address. Also access on the web and see what it says about it’s hostname as opposed to when we used the host network

Now let us pull the nginx image and deploy it while noting some things

docker pull nginx

docker inspect nginx #note the default port for deploying nginx (80)

docker run --network host -d --name webserver nginx

docker ps #verify if nginx is running, shouldn’t be of course

docker ps -a # to list all running and exited containers, we should see nginx exited

docker logs webserver # we can see that it failed to start cos address was already in use

The none/null network in Docker

In Docker, the "none" or "null" network refers to a network mode where a container is not connected to any network. When you run a Docker container without specifying a network, it will use this null network mode by default.

Here's what happens when a container is launched with the null network:

1. **Isolation**: The container will be completely isolated from any network communication. It won't have access to the internet or any other containers' networks.
2. **Local Communication**: While the container won't have access to external networks, it can still communicate with processes running within the same container through localhost or loopback interface (127.0.0.1).
3. **Limited Use Cases**: Containers using the null network mode are typically used for scenarios where network connectivity is not required or is intentionally restricted. For instance, it might be used for batch processing, where the container only needs to perform computations without interacting with external systems.
4. **Security Considerations**: Using the null network mode can enhance security by reducing the attack surface of the container. If a container doesn't need network access, connecting it to the null network prevents any potential security vulnerabilities associated with network communication.

Docker 5

Docker Volumes

**Recap of previous knowledge**

**Note**:

We use Docker for containerization

We use Kubernetes for container orchestration/MGT

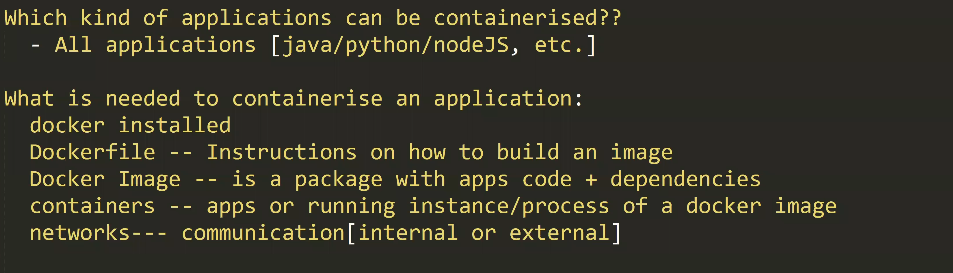
We can install the docker containerization software in Linux/Windows/MacOS, and when we do, we have access to;

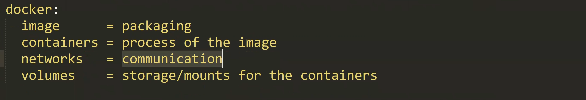
✅docker client /cli = docker

✅docker deamon/service =

✅docker registry = ship/share

We can use docker for Monolithic and Microservices applications





Now we shall see how very important Docker Volumes are when it comes to deploying stateful applications. So;

What are Docker Volumes?

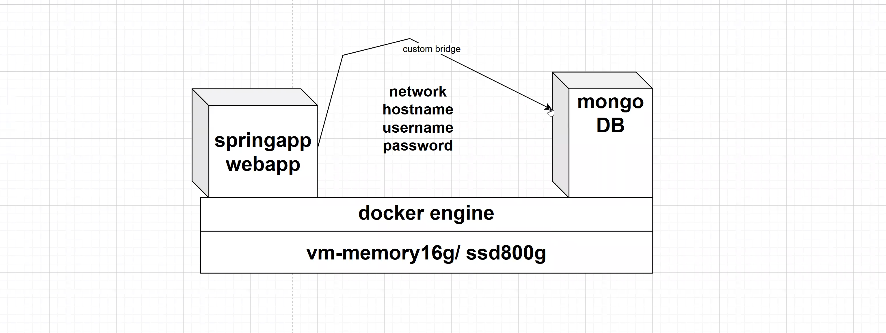
Docker volumes are a way to persist data generated by and used by Docker containers. When a container is deleted, any data created within it is also typically deleted. Volumes provide a solution to this problem by allowing data to persist beyond the lifetime of a container.

There are two main types of volumes in Docker:

1. **Named Volumes**: These are managed by Docker and can be easily managed and reused across different containers. They are stored within Docker's storage directory (**/var/lib/docker/volumes** on Linux). Named volumes are often preferred for production use because they are independent of the container's lifecycle.
2. **Host-mounted Volumes**: With host-mounted volumes, you specify a directory on the host machine that will be mounted into the container. This allows data to be shared between the host and the container. Host-mounted volumes can be useful for development or when you need direct access to the data from the host machine.

Volumes can be used for various purposes, such as storing configuration files, databases, logs, or any other data that needs to persist beyond the life of a container. They also facilitate data sharing and synchronization between containers.

Below is a typical scenario of how applications are deployed in docker, to communicate to a database, they need a network with environmental variables set correctly.

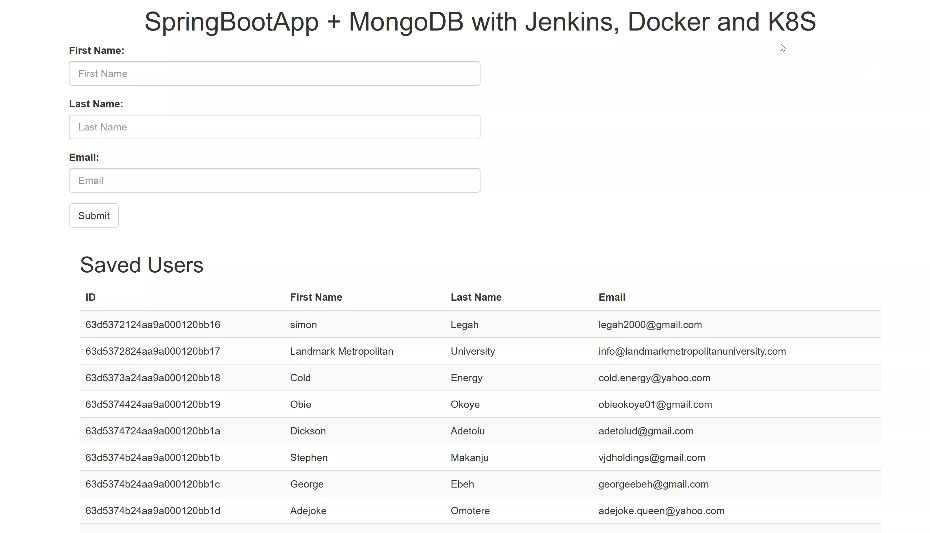


Back to the cli, if we run;

docker exec mongo ls data/db

we can see the data that the spring-boot-app has been saving in the mongo database.

Let us access the SpringBootApp on port 400 on the web and appreciate the data available like below;



So what if we removed or deleted the mongo container (our database);

docker rm -f mongo

Now if we refresh the web, we realize that all data won’t show anymore.

✅Now let us re-create the mongo container in the same network and see what happens

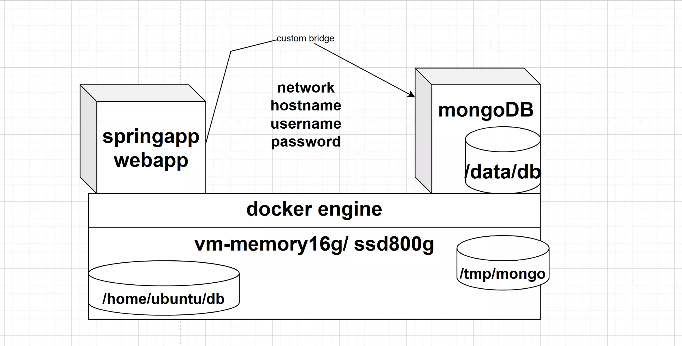
docker run –name mongo -d –network tesla -e MONGO\_INITDB\_ROOT\_PASSWORD=dev@123 -e MONGO\_NITDB\_ROOT\_USERNAME=devdb mongo

docker ps # to see that the container is up and running

if we refresh our app on the web still, we realize that we lost that previous data alongside the other instance of mongo db that we deleted.

So how do we solve this problem?

We can create a mount point anywhere on our host server, as depicted by the illustration below;



Let’s get back to our cli and start by removing the mongo db and then deploying a new one;

docker rm -f mongo

Now we intent to mount a new volume at /tmp/mongo on our host server, so let’s run the modified command as below;

docker run –name mongo -d –network tesla -v /tmp/mongo:/data/db -e MONGO\_INITDB\_ROOT\_PASSWORD=dev@123 -e MONGO\_NITDB\_ROOT\_USERNAME=devdb mongo

**Note** that, even if /tmp/mongo directory does not exist, docker will create it.

Having run above command, if you now do;

ls /tmp/ # we can appreciate that mongo folder has been created

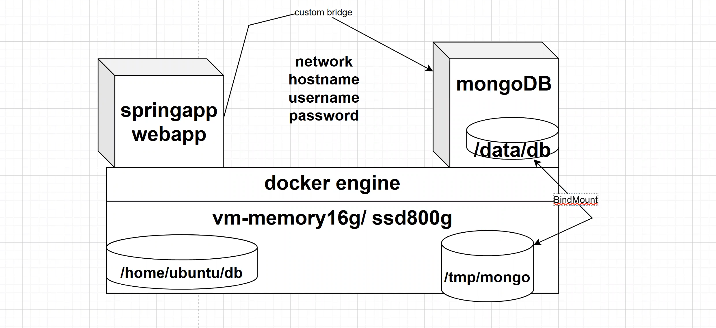
ls /tmp/mongo/ # we can see some default files created in the mongo folder

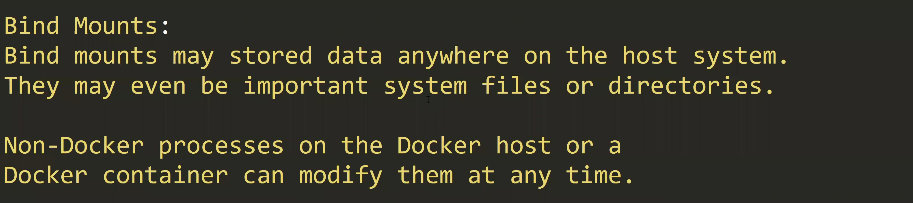
✅We also can retain that data stored in the container at /data/db is synced with that at /tmp/mongo. We can populate the mongo app to verify this.

✅If our database or mongo in this case is down or destroyed for any reason, we will still retain our data, and when we mount back our database, we will be able to have our data back up and running.

✅ We can test this by executing *docker rm -f mongo* again and then re-creating it back to see that our previously saved data is still available.

✅We call this kind of volume, the Bind Mount Volume;

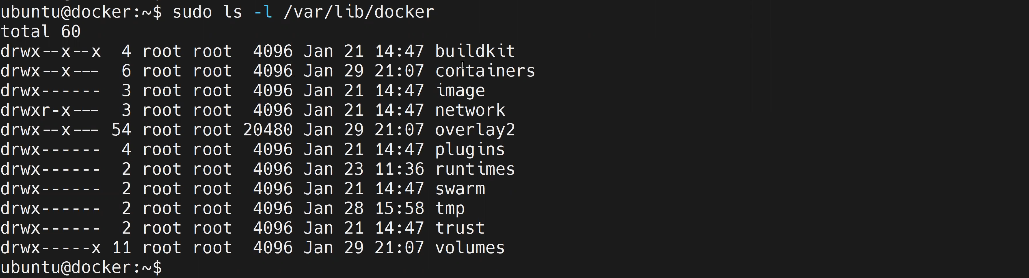




✅As opposed to Bind Mount Volumes, Docker Volumes store data in a part of the host filesystem managed by docker only. i.e in the docker home directory (DHD)



If we do ls -l /var/lib/docker, we should see a list of content like below;



So here we shall be talking about ;

Docker Persistent Volumes (Docker Named Volumes)

Command;

docker volume create mongodata # our volume is named mongodata

docker volume ls # list the existing volumes to see mongodata

docker volume create -d local data # create another volume data, in the local driver

docker volume inspect data # see the details, note the mount point

sudo ls -l /var/lib/docker/volumes/mongodata/\_data # folder should be empty for now

The folder above should be empty since deployment hasn’t been done using it, so let’s delete the running docker container and create another to test.

docker rm -f mongo

# after this verify that the spring boot app on web is still running and notice no data saved

# now the command below is used to create a container that will use docker volume (*data* as we created above) to save data on host server.

docker run –name mongo -d –network tesla -v data:/data/db -e MONGO\_INITDB\_ROOT\_PASSWORD=dev@123 -e MONGO\_NITDB\_ROOT\_USERNAME=devdb mongo

**Note**: if the docker volume *data* as specified in the above command did not exist, it will be created automatically.

Now ensure to populate the spring boot app with data then run the command below to check that the data is synced with the volume.

sudo ls -l /var/lib/docker/volumes/data/\_data # should contain data at this point

✅Hence, docker volumes are the best way to persist data even when the database container is deleted or fails for some reason.

**Question** 🤔: What if our server (ubuntu in this case) goes down? Won’t we therefore still lose our persistent data?

**Ans**: of course, yes, we would!

**Solution**: We can use EBS for external storage

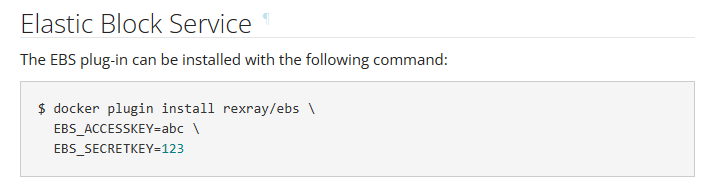
This brings us to the concept of ;

External Volumes

We can create an EBS volume and attach it to our network.

We can install the following plugin: Docker Rexray plugin for storage

<https://rexray.readthedocs.io/en/v0.9.1/user-guide/docker-plugins/>



Note: if your already have and IAM role in your docker server, you don’t need the access keys again.

On the command line, let’s do

docker plugin ls

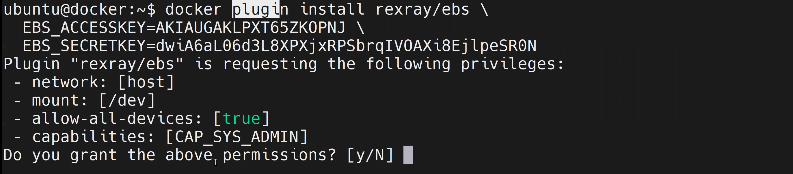
At first no plugin is available, (verify the volumes on aws account too by going to ec2 instances and clicking on volumes)so let’s then proceed to install the plugin

docker plugin install rexray/ebs \

EBS\_ACCESSKEY=abc \

EBS\_SECRETKEY=123

This plugin will do the mounting for us, something similar to this



We take yes and continue to install the plugin, when done, do;

docker plugin ls # we should see the plugin running

We can now use the plugin to create our external volumes.

If we do;

docker volume create ebs30 # this will create a volume with default driver ‘local’ and default network bridge.

So let’s rather do

docker volume create -d rexray/ebs ebs30 # we can verify the volume has been created in our aws account.

As mentioned before, let us run the volume create command like below;

docker volume create ebs40 #this should create for us a volume, but locally

We can’t find this volume under aws volumes, but if we do

docker volume ls # we should be able to see it, note that the driver is Local

# now let us remove our mongo container by running

docker rm -f mongo # refresh the springboot app to verify that no data is available

✅Let us deploy the MongoDB container using external volumes

docker run –name mongo d –network tesla \

-v ebs30:/data/db \

-e MONGO INITDB ROOT PASSWORD-dev@123 \

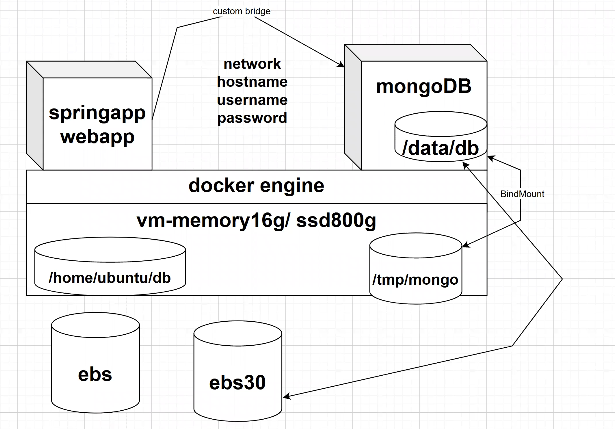
-e MONGO INITDB ROOT USERNAME-devdb mongo

#this mounting will take a longer time than when we are mounting our volume locally

docker ps #to check that the volume is mounted successfully

docker inspect mongo #to appreciate the details of the container

Diagrammatically, this is what we have now;



To appreciate this further, refresh the aws volumes page and notice that the state of the ebs30 volume is ‘in use’.

#Let us remove the mongo container again,

docker rm -f mongo #then verify the volume state once again, it should be ‘Available’.

✅Now, to appreciate the fact that we won’t lose our data when we destroy or delete our db containter, let us recreate it once again.

docker run –name mongo d –network tesla \

-v ebs30:/data/db \

-e MONGO INITDB ROOT PASSWORD-dev@123 \

-e MONGO INITDB ROOT USERNAME-devdb mongo

# Refresh the springboot app online to see that all the data is still intact, even though we had destroyed before recreating the database container.

✅Note that, by default, volumes are mounted as Read/Write, We can map volumes as Read Only using the below option.

-v <volumeName/BindMount>:<containerPath>:ro

This means that the above command should have been as below if we wanted to mount as Read Only.

docker run –name mongo d –network tesla \

-v ebs30:/data/db:ro \

-e MONGO INITDB ROOT PASSWORD-dev@123 \

-e MONGO INITDB ROOT USERNAME-devdb mongo

Docker Compose

Deploying applications the way we have been doing above involves so many commands and environmental variables such that it is prone to many potential errors.

Docker compose comes in to ease the deployment issues we might face with the way we have done the deployments above.

✨Definition: Docker compose is a tool for defining/declaring and running multiple containerized micro-services applications.

In real time, one application can have more than 7 micro-services: e.g. ebay.com:

An e-commerce java based web application for ebay may look like below;

ebay-web-application:

Login

Registration

cart

Checkout

Payment

Order

mysql db

So, 6 micro-services plus a MySQL database.

Hence, we may have to run 7 docker run commands to accomplish this, which becomes really challenging. And in addition, we have to create volumes for the database.

✅ Good news is, we can resolve this by making use of docker compose

✅ We can make use of the **docker-compose.yml** (docker compose file default name).

✅ To create a docker-compose file, we need to know the version of docker we are running, we can do this by running;

docker –version # on our cli, to see the version

(We shall use compose version 3.1, which runs with docker 1.13.1+)

A docker compose file content might look something like this;

version: ‘3.1’

services:

springapp:

image: mylandmarktech/spring-boot-mongo

restart: always

ports:

- 3000:8080

networks:

- amazon

environment:

- MONGO\_DB\_HOSTNAME=mongo

- MONGO\_DB\_USERNAME=proddb

- MONGO\_DB\_PASSWORD=prod@123

mongo:

image: mongo

restart: always

volumes:

- data28:/data/db

networks:

- amazon

environment:

- MONGO\_DB\_USERNAME=proddb

- MONGO\_DB\_PASSWORD=prod@123

volumes:

data28:

driver: local

networks:

amazon:

driver: bridge

✨ Ensure to **vi** and create a file in your docker server, then paste the script above in.

vi docker-compose #creates a new docker-compose file.

There are some few docker compose commands that we can use for deployment;

sudo apt install docker-compose # to install docker compose in order to run compose commands

docker-compose config # verifies if the docker-compose file content is okay

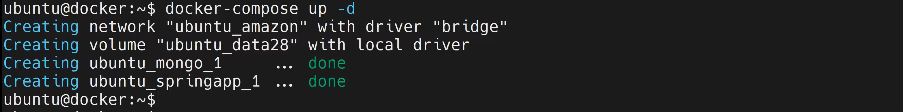
# you can mess up with the content of docker-compose.yml and see for yourself how docker-compose config command will through an error.

docker ps # to see the running containers

docker rm -f springapp mongo #or any other container name that is running, to remove them

docker-compose up -d #runs our docker-compose file, -d is for detachable

You should have an output like below if successful;



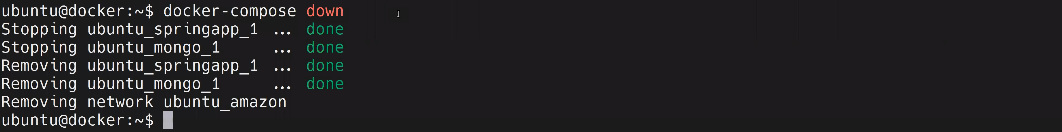
docker ps # to see that the applications are running...

👉 Now try accessing the application on port 3000 and you should be able to do just that.

👉 Try populating the application with data, and you should be able to do just that.

To delete or stop the applications from running, you run the command;

docker compose down #the opposite of the up command



✅ Deploying the applications with an existing network and volume, we use the keywords: external: true, as seen in the modified docker-compose file content below(highlighted are the changes made);

vi docker-compose.yml # to modify the file as required

docker network ls # to verify the available networks, in this case, ebay is available

# Also remember to verify the available volumes in your aws, chose one that is not in use, i.e. available.

version: ‘3.1’

services:

springapp:

image: mylandmarktech/spring-boot-mongo

restart: always

ports:

- 3000:8080

networks:

- ebay

environment:

- MONGO\_DB\_HOSTNAME=mongo

- MONGO\_DB\_USERNAME=proddb

- MONGO\_DB\_PASSWORD=prod@123

mongo:

image: mongo

restart: always

volumes:

- mongodata2:/data/db

networks:

- amazon

environment:

- MONGO\_DB\_USERNAME=proddb

- MONGO\_DB\_PASSWORD=prod@123

volumes:

mongodata2:

driver: rexray/ebs

networks:

ebay:

external: true

✅ Always remember to check that the file has no errors by running

docker-compose config

docker-compose up -d # takes a while longer to complete cos it’s using external network and volume

docker ps # to see the running containers

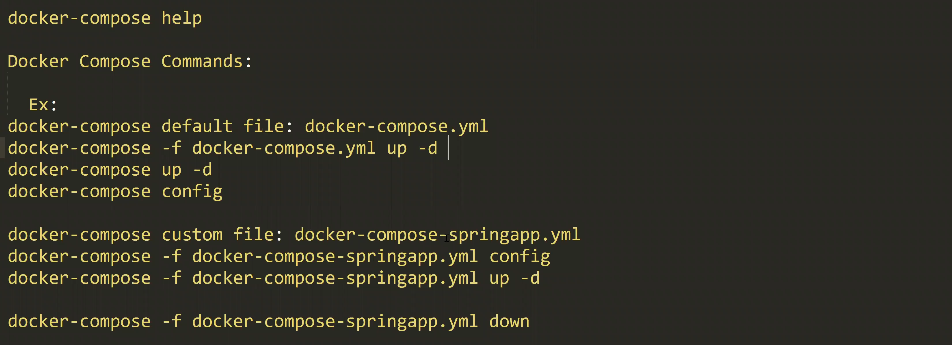
docker system prune # removes all stopped containers and networks not in use

Note that: if your docker-compose filename is not default, say dc.yml, you will have to run it like this:

docker-compose -f dc.yml up -d # for it to work

docker ps -a # to see all containers now available in the system

Some docker-compose commands for default file name and custom files names below:



**Docker Swarm** – will not be discussed, but is also used for container orchestration like Kubernetes. Resources will be given, notes and a video for the students.

We shall cover **Kubernetes** next, as it is a core part of the syllabus.

**IQ**: What is your experience in Docker?

**Ans**:

Thank you again, in our environment, we use Docker for containerization, as a DevOps Engineer, my task has to do with writing and modifying Docker files. Most of the time I modify existing Docker files to reflect our changes and updates. Once modified, I use them to package our applications and once packaged, I ship them to our image registry in DockerHub, from there we use Kubernetes to deploy our applications.

**Q**: Why don’t you use docker to deploy your applications?

**Ans**: This is because Docker does not support the overlay/multi-host network. Meaning we cannot create a cluster of different servers with running apps and they are able to communicate over a network.

To visualize, the following image shows such servers with apps running on them, but it is not supported in Docker:

