|  |  |
| --- | --- |
| **Docker** | 1. **Docker Overview**    1. **Docker Image**    2. **Docker Container**    3. **Docker network**    4. **Docker volume**    5. **Docker Register** 2. **Docker file** 3. **Docker compose**  * **Microservice with docker file /docker compose**  1. **Docker swarm (do youself – turorial is mentioned on reference link)** 2. **Reference**   **Documented by : Anil Gupta** |

**\*A Docker Overview**

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

**The Docker platform**

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host. You can easily share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Docker provides tooling and a platform to manage the lifecycle of your containers:

* Develop your application and its supporting components using containers.
* The container becomes the unit for distributing and testing your application.
* When you’re ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

**Fast, consistent delivery of your applications**

Docker streamlines the development lifecycle by allowing developers to work in standardized environments using local containers which provide your applications and services. Containers are great for continuous integration and continuous delivery (CI/CD) workflows.

Consider the following example scenario:

* Your developers write code locally and share their work with their colleagues using Docker containers.
* They use Docker to push their applications into a test environment and execute automated and manual tests.
* When developers find bugs, they can fix them in the development environment and redeploy them to the test environment for testing and validation.
* When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.

**Responsive deployment and scaling**

Docker’s container-based platform allows for highly portable workloads. Docker containers can run on a developer’s local laptop, on physical or virtual machines in a data center, on cloud providers, or in a mixture of environments.

Docker’s portability and lightweight nature also make it easy to dynamically manage workloads, scaling up or tearing down applications and services as business needs dictate, in near real time.

**Running more workloads on the same hardware**

Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines, so you can use more of your compute capacity to achieve your business goals. Docker is perfect for high density environments and for small and medium deployments where you need to do more with fewer resources.

**Docker architecture**

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

**The Docker daemon**

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

**The Docker client**

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

**Docker registries**

A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

**Docker objects**

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

**\*1.Docker IMAGES**

An *image* is a read-only template with instructions for creating a Docker container. Often, an image is *based on* another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a *Dockerfile* with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

|  |  |  |
| --- | --- | --- |
| sno | **[IMAGE]** Command | description |
| 1 | Docker image ls / Docker images 🡪list all docker images | |
| 2 | Docker image ls –format ‘{{.ID}} , {{.Repository}}’ 🡪output in comma separated format | |
| 3 | Docker image history [image id] | |
| 4 | Docker image rm -f [images id/names] 🡪 remove image –f ( if any container associated with this image force to remove) | |
| 5 | Docker image inspect [image name/image id] | |
| 6 | Docker image prune 🡪**[ it will remove all unused images , use carefully ]** | |
| 7 | **Save image into tar file**  Docker image save [image ] >[image name].tar 🡪 [first login as a super/root user] | |
| 8 | **Load tar file image**  **Docker image load < [image tar file]** | |
| 9 | **Docker image save VS docker container export**  **Save 🡪 it will contains all info of an image like tag, versions,layers etc**  ( docker image save ubuntu:16.04 ) -> only 16.04 version of tar file will be saved in one tar file  (docker image save ubunt) -> it will save all versions of ubuntu images in one tar file  **Export 🡪it will create image tar from running container (it will contains only basic info)** | |
| 10 | **Docker image load Vs docker image import**  Load -> it will contain all info of an image  Import -> it will contain only basic info of an image | |
|  |  | |

**\*2. DOCKER CONTAINERS**

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container’s network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

1. Container

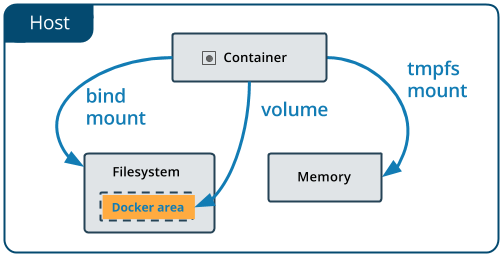
|  |  |  |
| --- | --- | --- |
|  | Command (moduler) | Desc |
| 1 | docker image ls | List of docker images |
| 2 | docker container ls | List running container |
| 3 | docker container ls -a | List all container running/stoped |
| 4 | docker container run [image name ]  or  docker container run ubuntu  [image name]=/ubuntu/linux/nginx etc  Or  Docker container run ubuntu sleep 30 | Create and run container of [image type] os , it will stop after creation of container  Create and run container for 30 second |
| 5 | docker container rm [ container-id ] | Remove container |
| 6 | Docker container rm [c-id1 c-id2] | Remove multiple container ,id separated by space |
| 7 | docker container start [container - id] | Start container |
| 8 | docker container stop [container-id] | Stop container |
| 9 | Docker container run –d [image name] [command]  Docker container run –d ubuntu /bin/bash | Detach container (run container in back-ground) |
| 10 | Docker container –it [image] [command]  Ex: -it(interactive terminal)  Docker container run –it ubuntu /bin/bash | Go inside a container ,and work like a separate os, you can install software as well, to exit from container cell =>type  Exit , now you will be in main cell, after typing exit container will close. |
| 11 | Docker container run 0–it ubuntu /bin/bash  Now press CTRL+pq | To keep container running and exit from container cell  Press ctrl+pq |
| 12 | docker container ls –aq | List all docker container id |
| 13 | docker container rm $(docker container ls –aq) | Delete all docker container |
| 14 | docker container inspect [running id]  ex: (run container in background)  docker container run -d nginx  297ba92abf…..  Docker container inspect 297ba92abf….. | Inspect all info about running container ,port,ip…..  We can check container server is running or not in browser   * Add ipadress to browser |
| 15 | Docker container logs [container id] | It will show logs of running container |
| 16 | docker container top 297ba92abf44 | Show what are the processes running of the container ,  Container is not a vm , it only runs the processes on top of  Host OS.OS treat it as a separate process. |
| 17 | Docker container stats | It will show all running containers state,cpu uses,memories.. |
| 18 | docker container run -d -p 3600:80 --name test\_web nginx  docker container run –d –P –name Test\_web nginx  ----------------  note: -P will assign dynamic port(any available port)  ex:  netstat –nltp (to check running port in local ubuntu)  (PORT mapping)We can redirect local request to , any container server  -p 3600:80 => map local 3600 port with containers 80 port  --name => any specific name of container.  => hit browser <http://172.18.78.151:3600/>  172.18.78.151=>local ubuntu system ip and 3600 port    Ex: open browse and type localhost/ ipaddress :port | |
| 19 | Docker container exec –it [container-id] /bin/bash | Excecute /bin/bash cell into container / or get cell access of current container,  Now you can install softwares into containers |
| 20 | docker container rename [container –id ] [name] | Rename container |
| 21 | Docker container restart [container-id ] | Restart container |
| 22 | docker container kill [container- id] | Force stop container , or kill |
| 23 | docker container wait [container-id] | Waitting state , for stop container , if container will stop then it will leave docker cell with int (0) return msg. |
| 24 | Docker container pause [container-id] | State of container will change form running to pausesd,  Now we can not able to access from out side of browser. |
| 25 | Docker container unpause [container -id] | Unpause container |
| 26 | Docker container prune –f | Prune will delete , all container which is not in use.  Only running will remains (-f will not show warnings) |
| 27 | Docker container port [container –id/name] | Show container port mapping with parent OS. |
| 28 | Docker container create [image name] | It will create pull image and create container .  It will not start, during creation. |
| 29 | docker container diff [container-id]  watch ‘docker container diff [container id]’ | It will show the details of created, updated and deleted files and folders  Watch is linux command which is used to watch container activity in every 2.0s |
| 30 | **Export container**   * docker container run -it ubuntu /bin/bash #[ 🡪create container and inter in container cell ] * sudo apt-get update #[🡪 update container softwares ] * apt-get install tree git –y #[ 🡪 install tree and git in container ] * ctrl+pq #[🡪 exit cell with running container] * docker container ls * **docker container export [container –id ] >ubuntu\_g\_t.tar** * ls –lh 🡪 list files , and it will show create tar file | |
| 31 | **Import container (.tar) and run in docker**   * Docker image ls * **Docker image import ubuntu\_g\_t.tar [image name]** 🡪 Docker image import ubuntu\_g\_t.tar **imported\_ubuntu\_image** * Docker image ls 🡪 now it will contain imported\_ubuntu\_image * Docker container run –it imported\_ubuntu\_image /bin/bash 🡪 run imported ubuntu\_image and give cell * Git –version 🡪 check git is there or not | |
| 32 | **Creating an image from running container ( and stop and remove running container , start container with created image and test)**   * docker container run -i -t ubuntu /bin/bash 🡪create and run container give cell to container * cd /tmp/ 🡪 go to tmp directory * touch 1 2 3 4 5 6 🡪 create files on tmp * ls –lh 🡪 list all files of tmp * ctrl+pq 🡪 exit cell with running container * docker container ls 🡪list available running container * docker container commit --author "Anil Gupta" -m "this is test commit" c7a981d8784d image\_crt\_with\_running\_container🡪 create image from running container * docker container ls 🡪 list running container * docker container rm -f c7a981d8784d 🡪 stop and remove running container * docker container ls 🡪list running container (check removed or not) * docker container run -it image\_crt\_with\_running\_container /bin/bash 🡪 run container with created img * cd tmp/ 🡪 go to temp directory * ls –lh 🡪 check files are there or not | |
| 33 | **Pull and push docker images from dockerhub (default image path docker.io)**   * Search images in dockerhub (example **docker.io/**ubuntu 21.04 v) * **docker pull ubuntu:21.04** * docker run –it ubuntu:21.04 /bin/bash 🡪 run docker container with ubuntu\_image\_21.04 v * apt-get update * apt-get install tree git –y * ctrl+pq * docker container commit --author "Anil Gupta" -m "this is test commit" c7a981d8784d image\_ubuntu\_21.04\_tree\_git * docker image ls * docker image tag [ image\_name] [dockeruser]/[image\_name]   ex: docker image tag image\_ubuntu\_21.04\_tree\_git dockerhub825313/ubuntu\_21\_04\_tree\_git   * docker login 🡪 docker hub login credential username and password (msg- login succeeded) * docker container ls * **docker push dockerhub825313/ubuntu\_21\_04\_tree\_git** 🡪start pushing to dockerhub | |

**\*3. Docker Volume**

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While [bind mounts](https://docs.docker.com/storage/bind-mounts/) are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

* Volumes are easier to back up or migrate than bind mounts.
* You can manage volumes using Docker CLI commands or the Docker API.
* Volumes work on both Linux and Windows containers.
* Volumes can be more safely shared among multiple containers.
* Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
* New volumes can have their content pre-populated by a container.
* Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.

In addition, volumes are often a better choice than persisting data in a container’s writable layer, because a volume does not increase the size of the containers using it, and the volume’s contents exist outside the lifecycle of a given container.



|  |
| --- |
| **Docker (Volume)**   1. Docker volume create volumeTest; 2. Docker volume ls; 3. Docker volume rm [volume name] 4. Docker volume prune 🡪 [it will remove all unused volume 🡪 if volume is associated with container then if need to remove that container before remove ] 5. **docker volume** rm $(**docker volume** ls -q) 🡪remove all volumes 6. **docker volume inspect [ volume name ]** |

|  |
| --- |
| Dockerfile (**persist data using** **1.** **Volume/ 2. bind mount / 3.tmpfs mount**)  <https://docs.docker.com/storage/>   1. Docker volume ls      1. docker pull mysql      1. docker image inspect mysql (for mysql image docker will create volumes inside local var/lib/mysql after running container)      1. Docker container run –d –name mysql –e MYSQL\_ALLOW\_EMPTY\_PASSWORD=true mysql      1. Docker volume ls (every time new volume will be created for new container instance)      1. Docker volume inspect [volume name]      1. Docker container exec –it [container id ] bash (**go inside running container**)      1. go inside mysql      1. show databases ;      1. create database Test1 (and show databases)      1. exit and delete container      1. Docker container run –d –name mysql –e MYSQL\_ALLOW\_EMPTY\_PASSWORD=ture mysql   (create new container )     1. Docker volume ls (both instance has separate volume and new one will start with fresh)      1. Docker exec –it [container id] bash      1. mysql databases;      1. docker container inspect [container –id ] ->[inside mount, we will find volume name associated with container ]      1. **now** create new container and **attach previous volume** with image , and tell container to stop   creating new volume but use existing with previous existing data.  Docker container run –itd –v [volume1]:/var/lib/mysql mysql (if volume will not exist than it will create new volume with name volume1)  docker container run -itd –v [volume name]:/var/lib/mysql mysql     1. go inside container and check mysql databases   docker exec –it [container id] bash (type mysql)   1. show databases; (we can see associate volumes contains previous database records) |

|  |
| --- |
| **Bind mount (we can mount , host machine file and folder to container)**  **Share window file with linux**   1. create SharedFolder 🡪properties->sharing->share with everyone      1. sudo apt install cifs-utils      1. Sudo mkdir /share 🡪create forlder in linux      1. Check ip-address of window (wls)      1. sudo mount.cifs //192.168.25.113/folder-shared-with-linux /share -o user=anil      1. check     **2. create container with bind mount (shared folder as volume)**  1. **docker container run -it -v /share:/tmp/test/ ubuntu:21.04 bash**  [it will create test folder inside temp and mount share folder on test]  2. check (cd tmp/test) |

**\*4 Docker Network**

One of the reasons Docker containers and services are so powerful is that you can connect them together, or connect them to non-Docker workloads. Docker containers and services do not even need to be aware that they are deployed on Docker, or whether their peers are also Docker workloads or not. Whether your Docker hosts run Linux, Windows, or a mix of the two, you can use Docker to manage them in a platform-agnostic way.

This topic defines some basic Docker networking concepts and prepares you to design and deploy your applications to take full advantage of these capabilities.

## Network drivers[🔗](https://docs.docker.com/network/#network-drivers)

Docker’s networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

* bridge: The default network driver. If you don’t specify a driver, this is the type of network you are creating. **Bridge networks are usually used when your applications run in standalone containers that need to communicate.** See [bridge networks](https://docs.docker.com/network/bridge/).
* host: For standalone containers, remove network isolation between the container and the Docker host, and use the host’s networking directly. See [use the host network](https://docs.docker.com/network/host/).
* overlay: Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers. See [overlay networks](https://docs.docker.com/network/overlay/).
* macvlan: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host’s network stack. See [Macvlan networks](https://docs.docker.com/network/macvlan/).
* none: For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services. See [disable container networking](https://docs.docker.com/network/none/).
* [Network plugins](https://docs.docker.com/engine/extend/plugins_services/): You can install and use third-party network plugins with Docker. These plugins are available from [Docker Hub](https://hub.docker.com/search?category=network&q=&type=plugin) or from third-party vendors. See the vendor’s documentation for installing and using a given network plugin.

**Network driver summary**

* **User-defined bridge networks** are best when you need multiple containers to communicate on the same Docker host.
* **Host networks** are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.
* **Overlay networks** are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
* **Macvlan networks** are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.
* **Third-party network plugins** allow you to integrate Docker with specialized network stacks.

|  |
| --- |
| **Docker Network**   1. **default docker network**      1. network commamd      1. **if there will be no container running , network will be free. (check with inspect)**   **docker network inpect [network id]**       1. **start at-least one container and inspect netwok bridge**   **docker container run –it nginx 🡪ctrl+pq🡪docker container ls**    **Docker network inspect [network id]**       1. **check ifconfig , virtual network list**      1. **docker network create –d bridge test (create “test”network of driver type bridge)**      1. attach network with container   **docker container run –it –-network test ubuntu:21.04 bash**     1. docker container inspect [container id] |

|  |
| --- |
| **Docker network demo**     1. create docker container with default network (bridge)   docker container run -d -p 8001:80 nginx  docker container run -d -p 8002:80 nginx  docker container run --network=test -d -p 8003:80 nginx  docker container run --network=test -d -p 8004:80 nginx       1. install necessary software for ip(net-tools),ping(iputils-tool),getfile(wget)   each container   * + **docker container exec –it [container-id] /bin/bash**   + **apt-get update**   + **apt-get install -y wget net-tools iputils-ping inside**  1. inspect bridge info (for network type and associated containers)     b> docker network inspect **55ccc5c1c702**    c)     1. check container ipAddress and ping (same network )      1. check wheather it will able to ping (cross network) 2. check other network port (docker 0)     B) try to ping docker0 (network container with test network container)     1. connection of different network via host IpAddress   wget 192.168.134.81:8004 |

|  |
| --- |
| **Docker network** (hostname mapping with network)   * if we will create container with default network , we have to use ipAddress to ping container in same network. * If we want to enable hostname as IpAddress we need to map network aliases with container id to do so, create our own network (like test) , bydefault custome network aliases will be maped with hostname.  1. Create custom network   **Docker network create test**   1. Run docker container in test network (two container)   **docker container run --network=test -it ubuntu:14.04 bash**  **docker container run --network=test -it ubuntu:14.04 bash**   1. Ping from one container to other container via hostname (you can able to ping successfully)      1. to check difference inspect containers |

|  |
| --- |
| **Docker network Host**  **\*. If we will run a container with host network then container will treat like a host system. you can access host container via their ipAddress directly (because both container and host will work on same ipaddress) bydefault iPAdress will point 80/tcp port.**   1. **Docker container run --network=host -itd nginx** 2. **Docker container exec –it [container id] /bin/bash** 3. **Apt-get update and apt-get install –y net-tools** 4. **Ifconfig**     **\*Above all machine have different port , no any host port matching (it is not working on my sys)**  **\*show I can not able to directly access container host via window host port (**[**http://192.168.134.81/**](http://192.168.134.81/)**)**   1. **Docker container inspect [container id]**     **Host container will not contain any IPaddress/gatway** |

|  |
| --- |
| Docker network none (null)  \*. By-default container has bridge network associated with container , if we don’t want to attach any network with container , we use **network type none**   1. Docker container run –network=none -it ubuntu:14.04 bash 2. Ifconfig      1. Docker container inspect [container -id] |

|  |
| --- |
| **Docker network (multiple network mapping with a container)**  **[network connect , network disconnect ]**   * Bydefault one network will ve associated with container (bridge), we can add more network of only brige type to an container. We can’t add multiple network on host and none network      1. **Docker container run –-network=bridge –it ubuntu:14.04 bash**      1. **docker network connect test f8cdc693c678**  [Add network test on existing cotainer ]      1. **docker network disconnect test f8cdc693c678**  [ remove network ] 2. **try to add other network** |

**\*5.Registry Server (unsecure)**

The Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images

## Why use it

You should use the Registry if you want to:

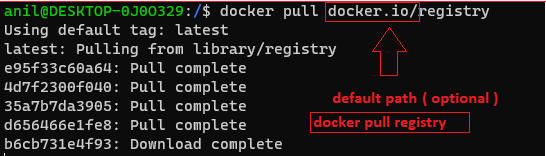
* tightly control where your images are being stored
* fully own your images distribution pipeline
* integrate image storage and distribution tightly into your in-house development workflow

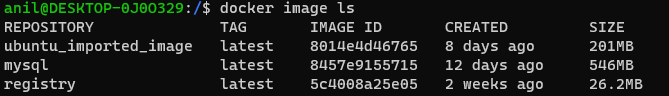
\*\* The Registry is compatible with Docker engine **version 1.6.0 or higher**.

**(unsecure registry server / pull and push only allowed via 127.0.0.1/8 or HTTPS url)**

* **In general we are using docker.io to pull and push the image.**
* **We need to create custom Registry to push images and pull images.**

1. **Docker pull registry**





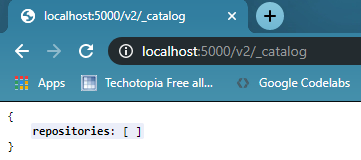
1. Create simple\_registry container from registry image with default volume

docker container run -it -d -p 5000:5000 --name simple\_registry registry



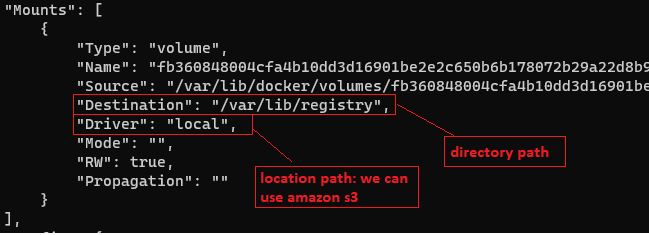
1. Check images in repository of private registry (simple\_resgisty)

127.0.0.1:5000/v2/\_catalog

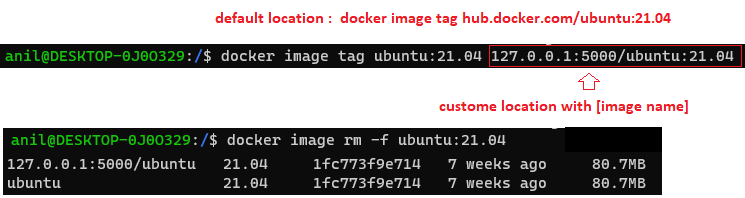


1. [customize mount path as per storage location/directory]

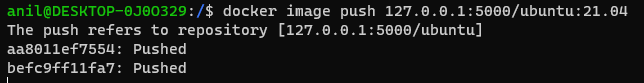
Docker container insptect [container id]



1. Docker image tag ubuntu:21.04 [url path]:[port]/[image name]



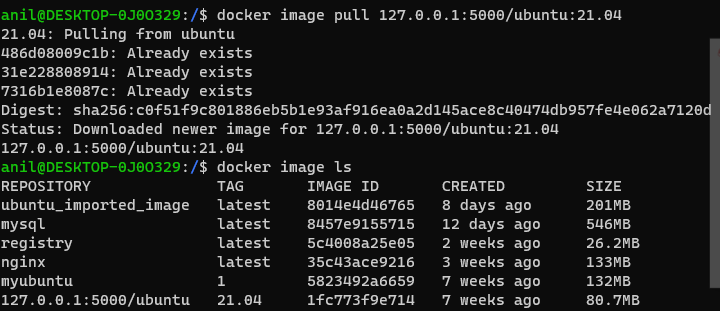
1. Docker image push 127.0.0.1:5000/ubuntu:21.04 [push in local registry server]



1. check



1. Pull docker image from local registry server



**Docker Registry Server (unsecure server)**

**( allow other url to push and pull images , other than 127.0.0.1 and https)**

1. **Check we can able to pull or push via other url path**

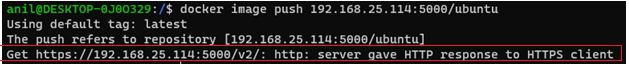
****

****

****

****

**docker image push 192.168.25.144:5000/ubuntu:14.04**

****

1. Resolved the issue by adding a file /etc/docker/daemon.json with following content



1. Restart docker
2. Docker image push 192.168.25.114:5000/ubuntu:14.04



(it should work but ,here not working --- we will check latter for this issue)

1. **Docker File**

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession.

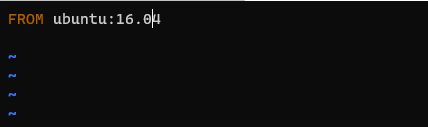
**1.Create Docker File (layed architecture)**

1. Vi Dockerfile 🡪 create/open Dockerfile (name Dockerfile is default , docker will search Dockerfile for build from path)

and press I (-- insert mode --)

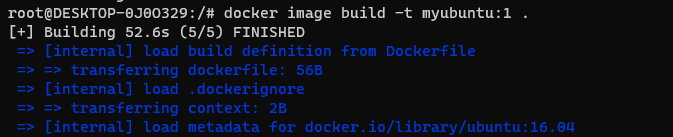


1. FROM ubuntu:16.04

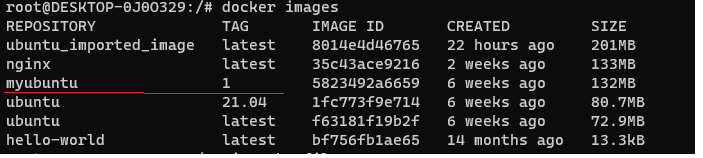


1. Press esc -> :w ( enter to save) -> :q (enter to save and exit cell)
2. Docker image build –t [give any name of image ] [path]

**docker image build -t myubuntu:1 .** 🡪(**.** Dockerfile is present in root directry, add tag for versioning)



1. Docker image ls



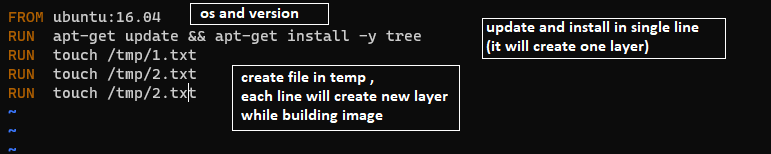
1. Create and run docker container using created image

Docker container –it myubuntu:1

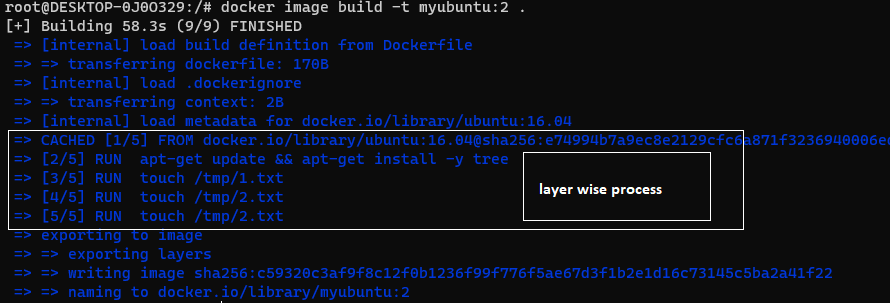


1. **Create Dockerfile (of OS ubuntu 16.04 and inatall software )**
2. **Vi Dockerfile and press I (-- insert mode --)**

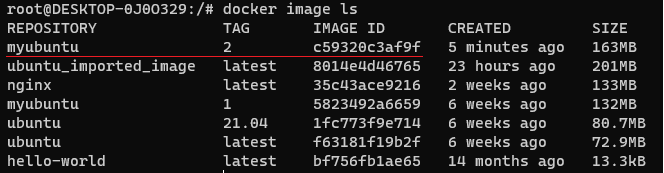
****



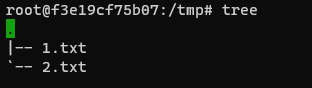
1. **Press esc -> :q(enter to save and exit )**
2. **Docker image build –t myubuntu:2 .**



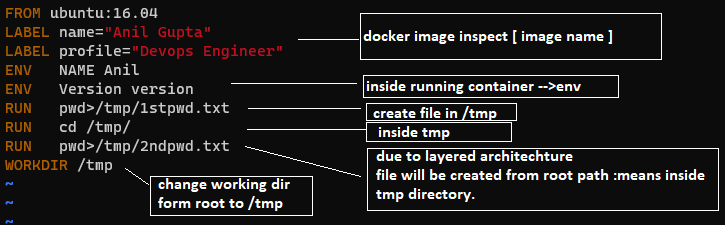
1. **Docker image ls**



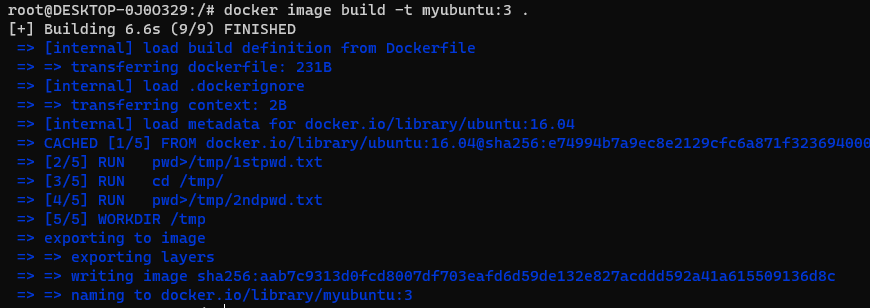
1. **Docker container run -it myubuntu:2 /bin/bash** (check tree and created file available or not)

****

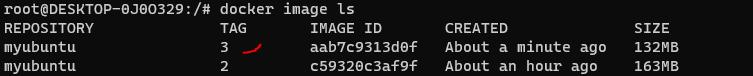
1. **Dockerfile (label, ,env ,run,workdir)**
2. Vi Dockerfile and press I (-- insert mode --)



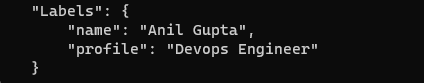
1. Press esc-> :q(save and exit cell)
2. Docker image build –t myubuntu:3 .

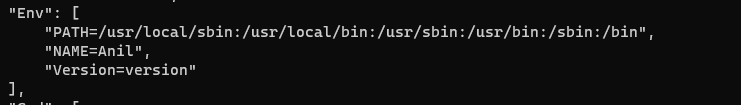


1. Docker image ls

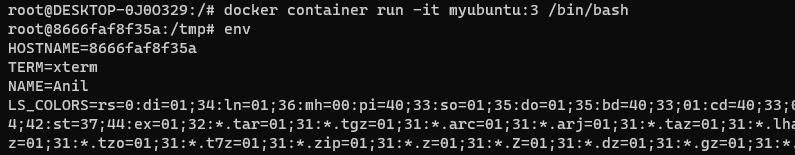


1. Docker image inspect myubuntu:3 (check level and env)

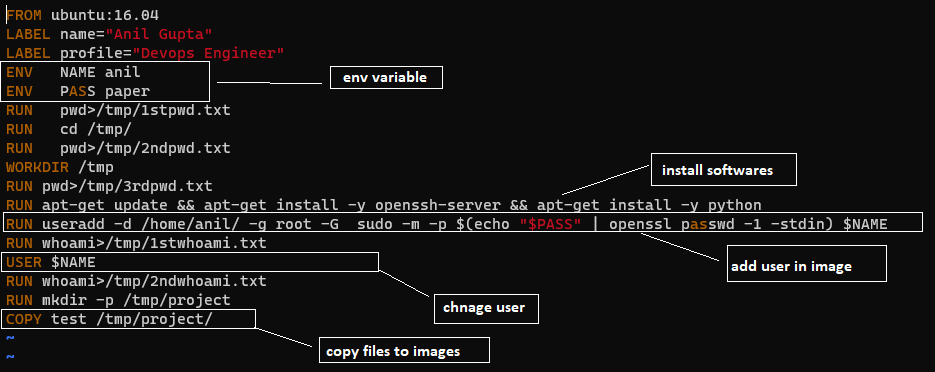




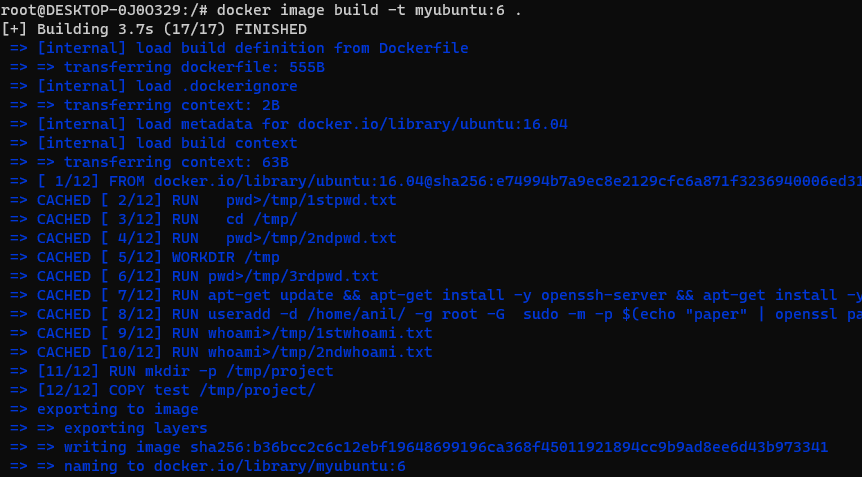
1. Docker container run –it myubuntu:3 /bin/bash (check env and temp for files)



1. **Dockerfile (create user inside image, copy (copy only files ) vs add (copy and extract file if tar file) , Add)**
2. Vi Dockerfile and press I for (--insert mode--)



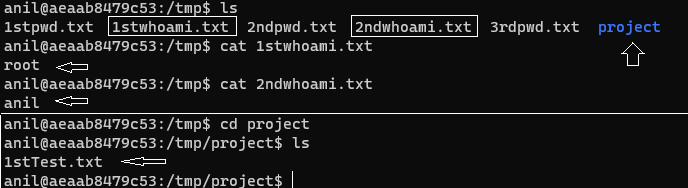
1. Press esc -> :wq (save and exit from cell)
2. Docker image build –t myubuntu:6 .



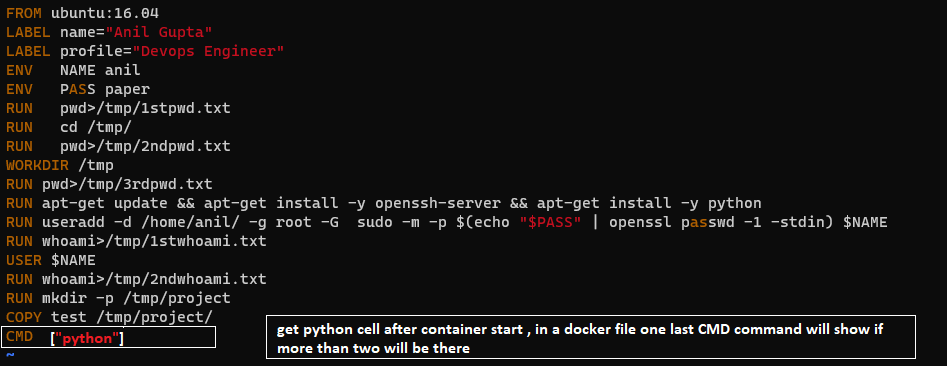
1. Docker image ls



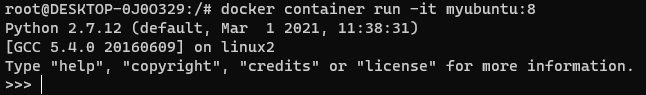
1. Docker container run –it myubuntu:6 /bin/bash



1. **Dockerfile (CMD, get particular cell after running container )**
2. vi Dockerfile and press I for (--insert mode--)

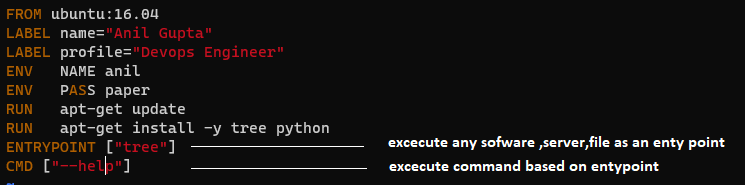


1. Press esc -> :wq(save and exit)
2. Docker image build –t myubuntu:8 .
3. Docker container run –it myubuntu:8

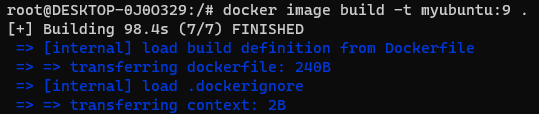


**6. Dockerfile (entry point -> set default entry point when container starts)**

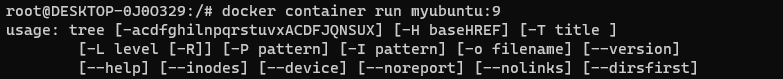
1. Vi Dockerfile and press I (-- insert mode --)



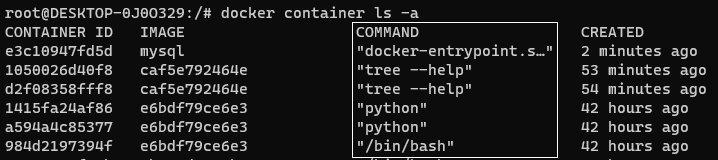
1. Press esc enter -> :wq (to exit cell)
2. Docker image build –t myubuntu:9 .



1. Docker image ls
2. Docker container run myubuntu:9



1. Docker container ls –a



1. **Docker Compose**

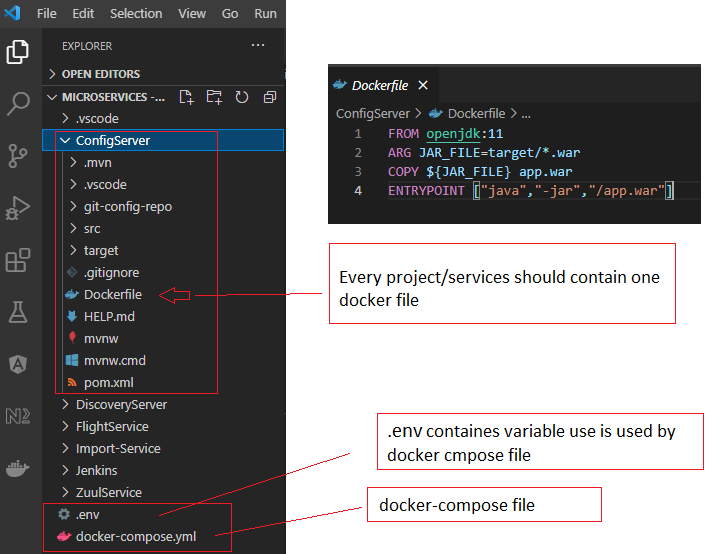
Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration. To learn more about all the features of Compose, see [the list of features](https://docs.docker.com/compose/#features).

Compose works in all environments: production, staging, development, testing, as well as CI workflows. You can learn more about each case in [Common Use Cases](https://docs.docker.com/compose/#common-use-cases).

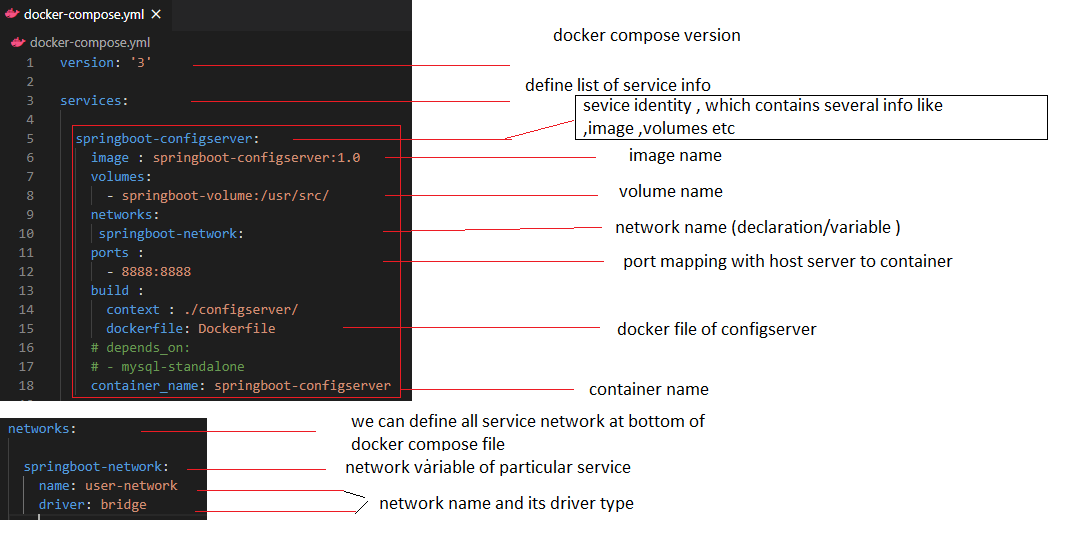
Using Compose is basically a three-step process:

1. Define your app’s environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Run docker compose up and the [Docker compose command](https://docs.docker.com/compose/cli-command/) starts and runs your entire app. You can alternatively run docker-compose up using the docker-compose binary.

**{ Microservice Project Architecture/ and package overview with docker }**

****

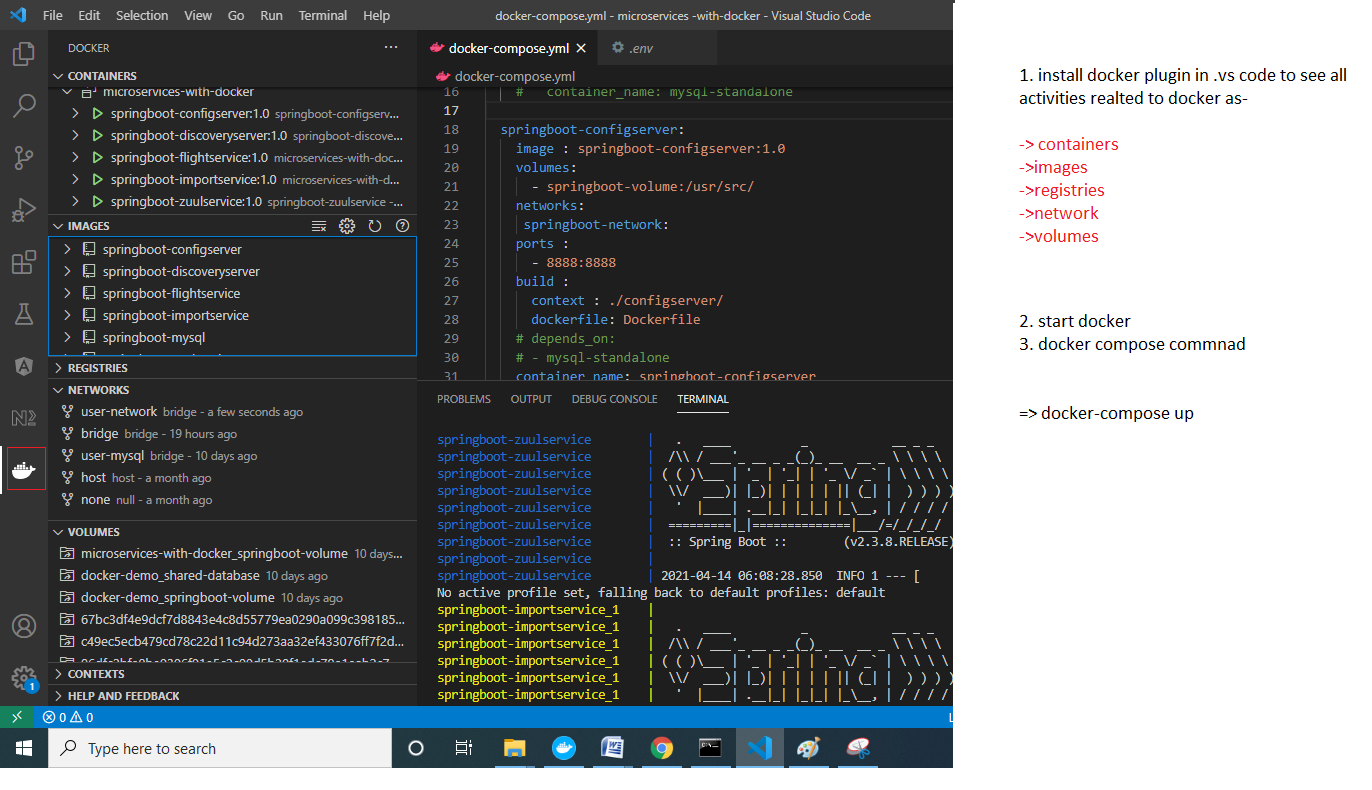
**Docker-compose.yml**

****

**Docker-compose with .env file**

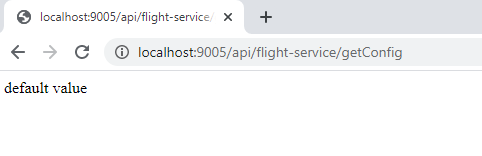
****

**Install docker plugin in vs code and run docker-compose command to start containers**

****

**Check server is running or not via browser (host machine based on port mapping)**

* + Below 9005 is related to zull api gateway which is running successfully

****

**Without docker compose (steps to create container of individual service )**

**1. mvn package -DskipTests // mvn -f ./import-service/pom.xml package -DskipTests**

**2. docker build -t springboot-mysql:1.0 . // read docker file and crate image**

**3. docker container run -it -p 9002:9002 --network=user-mysql springboot-mysql:1.0**

**4. docker logs [container id]**

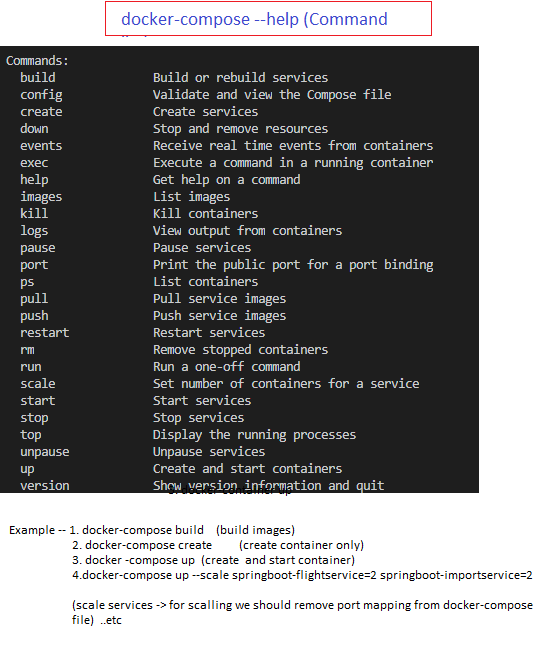
**Disable websecurity (or cors , add below code to crome properties )**

**"C:\Program Files\Google\Chrome\Application\chrome.exe" --disable-web-security --user-data-dir="C:/ChromeDevSession"**

**\*Access via browser (check api is working of not)**

**http://localhost:9002/api/flight-service/getImportService**

**Docker-compose commands :**

****

**Reference : -**

* 1. [**https://docs.docker.com/**](https://docs.docker.com/)
  2. [**https://www.youtube.com/watch?v=ETBj0oxe81o&list=PL6XT0grm\_Tfje2ySztzdhp0HmCjVj5P4z**](https://www.youtube.com/watch?v=ETBj0oxe81o&list=PL6XT0grm_Tfje2ySztzdhp0HmCjVj5P4z) **[gourav Sharma docker tutorial]**

**Thanks**