Contents

[Docker 3](#_Toc152841458)

[WHAT IS DOCKER? 3](#_Toc152841459)

[WHY DOCKER? 3](#_Toc152841460)

[USING DOCKER CLIENT? 3](#_Toc152841461)

[WHAT IS CONTAINER? 4](#_Toc152841462)

[RELATION BETWEEN CONTAINER AND IMAGE 5](#_Toc152841463)

[LINUX KERNEL 6](#_Toc152841464)

[DOCKER COMMANDS 6](#_Toc152841465)

[DOCKER RUN 6](#_Toc152841466)

[LIST ALL RUNNING CONTAINER 7](#_Toc152841467)

[STOPING THE RUNNING CONTAINER 7](#_Toc152841468)

[REMOVING THE CONTAINER 8](#_Toc152841469)

[INSPECT CONTAINER 8](#_Toc152841470)

[DOCKER IMAGES COMMAND 8](#_Toc152841471)

[EXECUTING A TASK IN A RUNNING CONTAINER 8](#_Toc152841472)

[EXAMPLES 8](#_Toc152841473)

[ATTACHED AND DETACHED MODE OF RUNNING CONTAINER 9](#_Toc152841474)

[ATTACHED MODE 9](#_Toc152841475)

[DETACHED MODE 9](#_Toc152841476)

[TAGGING 9](#_Toc152841477)

[CONTAINER IN STANDARD INPUT & OUTPUT 9](#_Toc152841478)

[RUNNING CONTAINER IN INTERACTIVE MODE 10](#_Toc152841479)

[CONTAINER - PORT MAPPING 10](#_Toc152841480)

[STEPS TO MAP THE PORT 10](#_Toc152841481)

[EXAMPLE 11](#_Toc152841482)

[DATA PERSISTENCE IN DOCKER CONTAINER 11](#_Toc152841483)

[CONTAINER LOGS 12](#_Toc152841484)

[CREATING A DOCKER IMAGE 12](#_Toc152841485)

[CREATING A DOCKER FILE 12](#_Toc152841486)

[DOCKERFILE IN DETAILS 12](#_Toc152841487)

[DOCKER FILE FORMAT 13](#_Toc152841488)

[RUNNING THE DOCKER FILE 13](#_Toc152841489)

[LAYERED ARCHITECTURE OF DOCKER BUILD 14](#_Toc152841490)

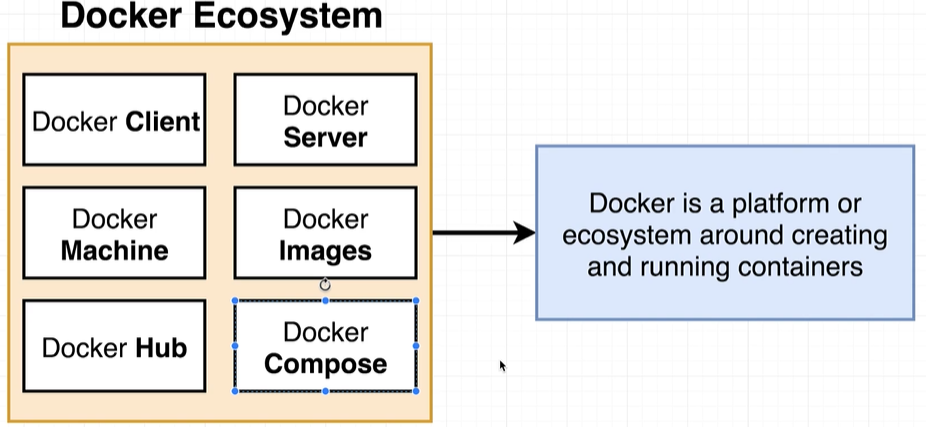
[ENVIRONMENT VARIABLES IN DOCKER 14](#_Toc152841491)

[COMMAND VERSUS ENTRYPOINT 15](#_Toc152841492)

[CREATING AN IMAGE IN LOCAL MACHINE 16](#_Toc152841493)

# Docker

## WHAT IS DOCKER?

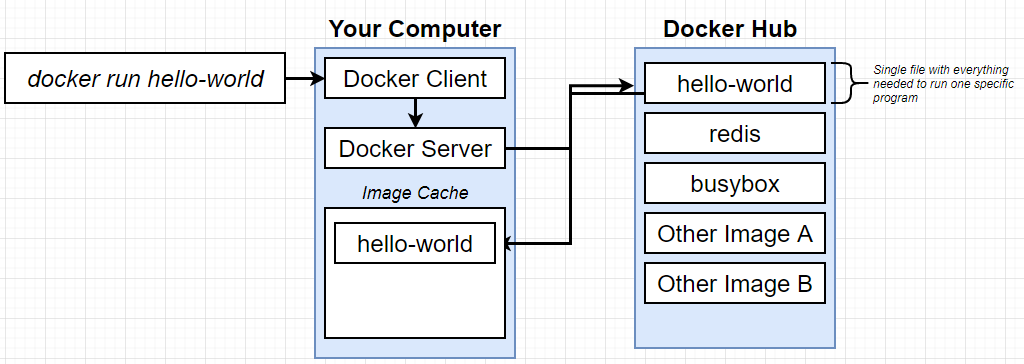


## WHY DOCKER?

* Docker makes it easy to install a software without worrying about set up or dependencies.

## USING DOCKER CLIENT?

|  |  |
| --- | --- |
| **DOCKER VERSION** | docker version |
| **DOWNLOADING AND RUNNING AN IMAGE IN A CONTAINER** | docker run *<image\_name>*  ***docker run hello-world*** |



* The purpose “docker run” command is to run the image in container.
* After we execute a “docker run” command in docker client or docker CLI, it pass the instruction to the Docker Server
* The docker first checks for the image in the local machine and if not found – it tries to download the image from a docker repo called “**Docker Hub”**
* Once it is downloaded from the docker hub – it stores it in image cache (in the local machine)- it avoids the re-downloading of the same image

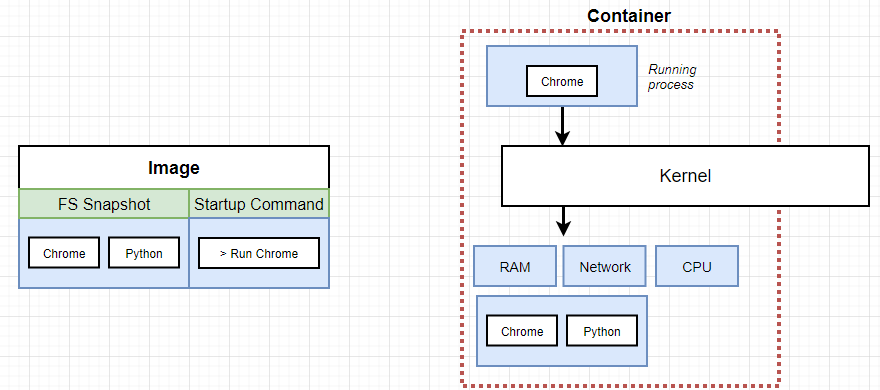
|  |  |
| --- | --- |
| * Once the image is downloaded – It starts a dedicated process called container and start executing the program.   Note: Container is an instance of the image |  |

## WHAT IS CONTAINER?

|  |  |
| --- | --- |
| * In operating system – if an application needs to talk to the physical hardware devices via Kernel. * These apps make system calls to kernel which in turn interact with hardware.   ***Kernel is a running program which governs access to the running application and the physical hardware. Means it decides/ governs – which application can access which hardware device.*** |  |
| * To understand container – Lets consider a hypothetical analogy. * Let’s say, we have Python v2 installed in our machine and we have two apps – Chrome & NodeJS – which needs Python V2 and Node V3. * Due to incompatible- Python version Node JS will not work as it needs Python v3 |  |
|  |  |
| * The work around for this problem – is to create segments in the hard-drive and install the respective python version is the segments * Now based on the system call kernel will redirect it to respective segment. * This process of segregating the hardware and software resources is called “NameSpacing” | |
| * With name-spacing we can be able to isolate a resource per process or group of process. * On the other hand – Control Group is the amount of resource a particular process can use. * The entire vertical in the diagram – of the running process and the segment in the Hard disk is called container. * Hence container is a process which has a set of resources assigned to it.   **CONTAINER ON HIGH LEVEL** |  |
|  |

### RELATION BETWEEN CONTAINER AND IMAGE

* Image is basically – is a snapshot of the file system(like folder)
* The image has a start up command.
* When we run the container – the image gest copied to the container and the start-up command starts an instance of an image. It will have very specific group of resources and hardware.



## LINUX KERNEL

* As the name-spacing and control group feature are only available in Linus OS
* In the window/Mac OS – When we install docker - which in turn install Linux Virtual Machine, which has a Linux Kernel – which facilitate name-spacing and control group. The Linux Kernel host the running process i.e. containers.

|  |  |
| --- | --- |
|  | ***The docker version commnd gives the detail of Linux VM*** |

## DOCKER COMMANDS

### DOCKER RUN

|  |  |
| --- | --- |
| * Docker run command creates and run a container from an image. |  |

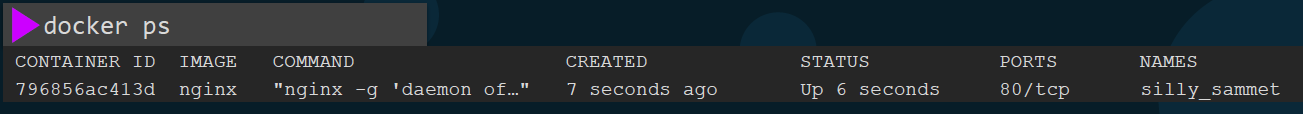
#### OVERRIDING DEFAULT COMMANDS

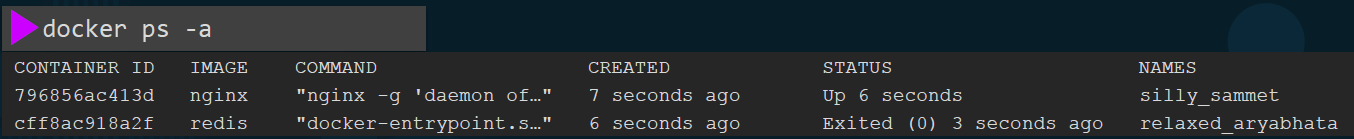
* “docker run” command by default create+ start the container. We can to override the default behavior of these commands. e.g.

|  |  |
| --- | --- |
| ***docker run busybox echo hi there*** | This will print “hi there” message in the command |
| ***docker run busybox echo ls*** | This will list the file system inside the dedicated container for this image |

### LIST ALL RUNNING CONTAINER

|  |  |
| --- | --- |
| **docker ps** | This will list all the running containers. |
| **docker ps --all** | To see all the containers – **both running and exited containers** |
| **docker start <CONTAINER\_ID>** | Starts a specific container. |

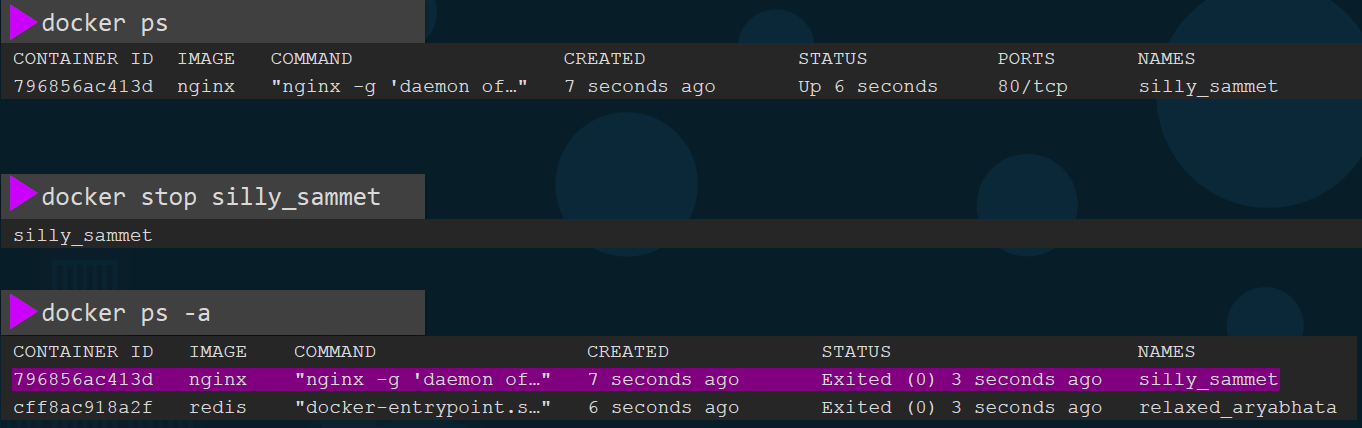




### STOPING THE RUNNING CONTAINER

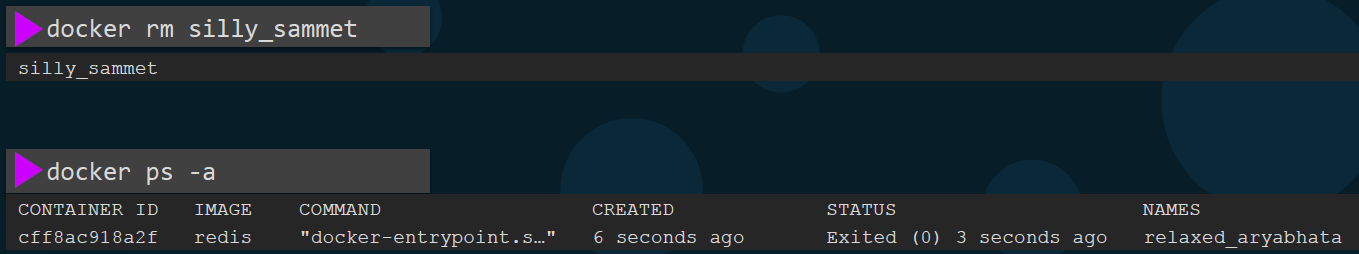
* Stopping the container will change the status from Running 🡪 Exited state.

1. **docker stop <container\_name>**
2. **docker stop <container\_id>**



### REMOVING THE CONTAINER

Removes the container

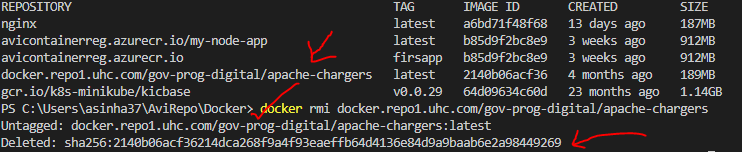


### INSPECT CONTAINER

|  |  |
| --- | --- |
| INSPECT THE IMAGE | docker inspect <container\_name>  docker inspect nginx |

### DOCKER IMAGES COMMAND

|  |  |
| --- | --- |
| COMMAND |  |
| docker images | List of all images and its sizes |
| docker rmi <Image\_Name>  docker rmi nginx | * Remove the specified image. * Before delete the image – we need to stop and delete all dependent container. |



|  |  |
| --- | --- |
| docker pull <image\_name>  docker pull nginx | * Only pulls that image and without running the container. * In the example - the command pulls the nginx image and store on our host. |

## EXECUTING A TASK IN A RUNNING CONTAINER

* The docker exec command is used to run a command inside a running container.
* SYNTAX: **docker exec [OPTIONS] CONTAINER COMMAND [ARG...]**

|  |  |
| --- | --- |
| OPTIONS | * Additional options that can be used with the docker exec command. * Some commonly used options include -i (interactive mode) and -t (allocate a pseudo-TTY). |
| CONTAINER | * The ID or name of the container in which you want to execute the command |
| COMMAND | * The command you want to run inside the container |
| [ARG...] | * Optional arguments to be passed to the command |

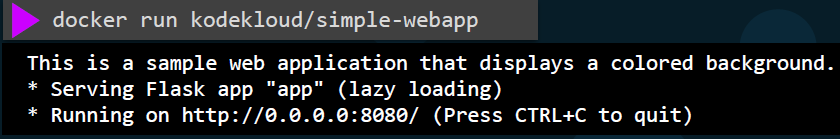
### EXAMPLES

|  |  |
| --- | --- |
| To run a shell command inside a container named "my-container” | docker exec -it my-container sh |
| Execute a single command inside a running container | docker exec my-container echo "Hello, world!" |
| Start an interactive shell session within a container | docker exec -it my-container bash |
| Execute a command as a different user within a container | docker exec -u <username> my-container command |
| Execute a command with environment variable(s) set within a container: | docker exec -e VAR\_NAME=value my-container command |
| View a file in a container | docker exec <container\_name> cat /etc/hosts |

## ATTACHED AND DETACHED MODE OF RUNNING CONTAINER

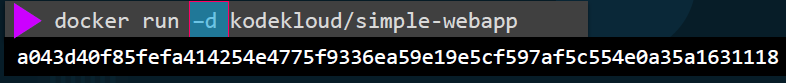
### ATTACHED MODE

* Let’s say we run “docker run” command (as shown below) which It runs a simple web server that listens on Port 8080.
* When we run a docker run command like this - It runs in the **foreground or in an attached mode** meaning we will be attached to the console or the standard out of the docker container and we will see the output of the web service on our screen.
* We won't be able to do anything else on this console, until this docker container stops (use CTRL + C to stop the container) then the application hosted on the container exits and we get back to our prompt



### DETACHED MODE

* We can run the docker container in the detached mode by providing the **-d** option.This will run the docker container in the background mode and we will be back to our prompt immediately.
* The container will continue to run in the backend run the Docker ps command to view the running container.

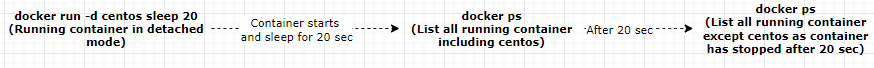


* To attach back to the running container later run the docker attach command and specify the name or I.D. of the docker container.

**NOTE**:



* if we are specifying the I.D. of a container in any Docker command we can simply provide the first few characters alone for example - **a043d** now.

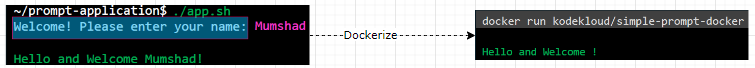


## TAGGING

* In Docker, image tagging is used to assign a specific name and version (tag) to an image.
* Tags are used to differentiate different versions or variants of an image.
* By default, when we pull an image from a registry, Docker uses the "latest" tag if no specific tag is specified.

|  |  |
| --- | --- |
| Pulling an image with a specific tag | **docker pull <image\_name>:<tag>** |
| Replace <image\_name> with the name of the image, and <tag> with the desired version or variant of the image | Example- to pull the "nginx" image with the "1.21.1" tag  **docker pull nginx:1.21.1** |
| Tagging an existing image | **docker tag <source\_image>:<source\_tag> <target\_image>:<target\_tag>** |
| Replace <source\_image> and <source\_tag> with the name and tag of the existing image, and <target\_image> and <target\_tag> with the desired name and tag for the new image | Example, to tag an existing image "myimage" with the "latest" tag:  **docker tag myimage:1.0 myimage:latest** |
| Pushing an image with a specific tag to a registry: | **docker push <image\_name>:<tag>** |
| Replace <image\_name> with the name of the image, and <tag> with the version or variant of the image you want to push | Example - To push an image "myimage" with the "1.0" tag to a registry:  **docker push myregistry/myimage:1.0** |

## CONTAINER IN STANDARD INPUT & OUTPUT



* Let’s say – We have a simple prompt application which prints message (with with name as input). Now if dockerize this application and run it as a docker container – it never prompt for the message. It just prints whatever the application is supposed to print on standard out.
* **That is because by default the Docker container does not listen to a standard input i.e. by default it runs in a non-interactive mode.**

### RUNNING CONTAINER IN INTERACTIVE MODE

* To run a Docker container in interactive mode, we can use the **-i (or --interactive) and -t (or --tty)** flags with the **docker run**command.
* This allows us to interact with the container's command line and see its output in real-time.

**SYNTAX** : docker run -it <image\_name> <command>

|  |  |
| --- | --- |
|  |  |

* -i or --interactive: Keep STDIN open and allow interaction with the container.
* -t or --tty: Allocate a pseudo-TTY for the container. The -t (or --tty) flag tells Docker to allocate a virtual terminal session within the container
* Replace <image\_name> with the name of the Docker image we want to run, and <command> with the command or entry point we want to execute within the container.

#### EXAMPLE

|  |  |
| --- | --- |
| TO RUN AN UBUNTU CONTAINER IN INTERACTIVE MODE AND START A SHELL SESSION INSIDE IT | docker run -it ubuntu /bin/bash   * This will launch a new container based on the Ubuntu image and open an interactive shell session within it. * We can run commands and interact with the container's shell just like we would on a regular command line. |
| IF WE WANT TO ATTACH TO A RUNNING CONTAINER AND ENTER INTERACTIVE MODE, We CAN USE THE docker exec COMMAND | For example, to attach to a running container named "my-container" in interactive mode:  docker exec -it my-container /bin/bash |

## CONTAINER - PORT MAPPING

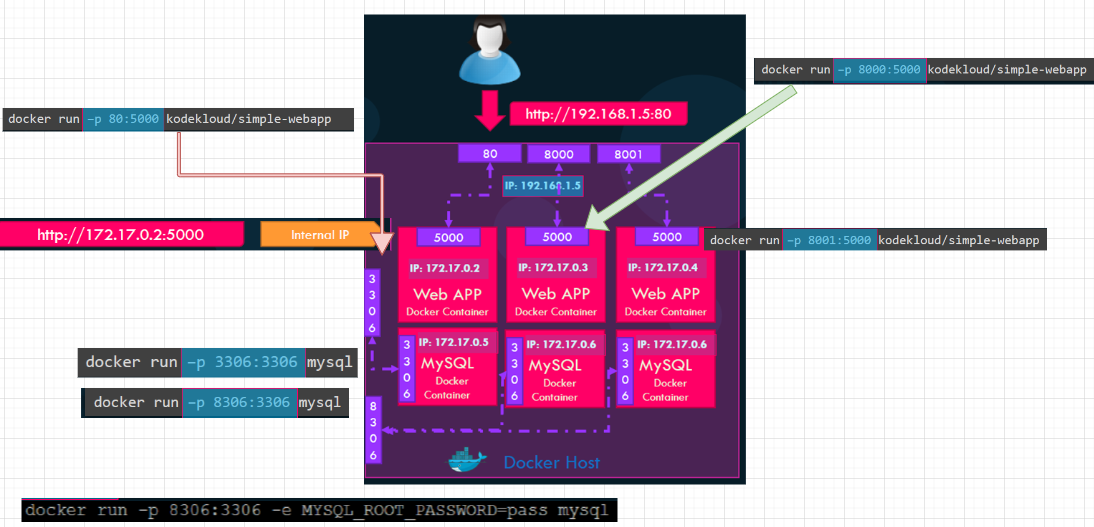
* Port mapping in Docker allows us to map network ports between the host machine and the container.
* This enables us to access services running inside a container from the host machine or other machines on the network.

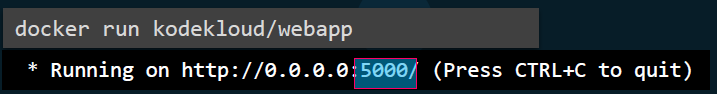
|  |  |
| --- | --- |
| * Use the -p (or --publish) flag with the docker run command * <host\_port>  : The port on the host machine that we want to map. * <container\_port> : The port on the container that we want to expose. | **SYNTAX**  **docker run -p <host\_port>:<container\_port> <image\_name>** |
| EXAMPLE:  To run an HTTP server in a container and map port 80 of the host machine to port 8080 inside the container. | **docker run -p 80:8080 <image\_name>** |
| We can specify the IP address of the host machine if we want to bind the container's port to a specific IP address | **docker run -p <host\_ip>:<host\_port>:<container\_port> <image\_name>**  *<host\_ip> is the IP address of the host machine* |
| We can use the -P (or --publish-all) flag to publish all exposed ports of the container to random ports on the host machine. | **docker run -P <image\_name>** |

### STEPS TO MAP THE PORT

|  |  |
| --- | --- |
| * Run the application in the container (nginx) * Inspect the container:   **docker inspect <container\_name>**  **docker inspect nginx**   * Check the Container configuration for exposed port (from the container) * Expose the port:   docker run -p 8080:80 nginx |  |

### EXAMPLE





PORT MAPPING:



* We can map port 80 of local host to port 5000 on the docker container using above command. Hence all traffic on port 80 on the docker host will get routed to port 5000 inside the Docker container.
* We can run multiple instances of the application and map them to different ports on the docker host or run instances of different applications on different ports.

## DATA PERSISTENCE IN DOCKER CONTAINER

* By default, any data written within a container is stored within the container's writable layer, which is ephemeral(transient) and gets deleted when the container is stopped or removed.

Docker provides several mechanisms to achieve data persistence:

**VOLUMES**:

|  |  |
| --- | --- |
|  | * Docker volumes are the recommended way to manage data persistence. * A volume is a specially designated directory within the Docker host's filesystem that can be mounted into one or more containers. * Volumes exist independently of containers and their data is preserved even if containers are stopped or removed. |

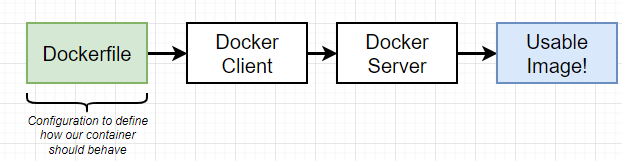
* Volumes can be managed using the docker volume command or specified in a Docker Compose file.

**SYNTAX: docker run -v <volume\_name>:<container\_path> <image\_name>**

## CONTAINER LOGS

|  |  |
| --- | --- |
| **TO VIEW LOGS OF THE CONTAINER** | * docker logs <container\_id> * docker logs <container\_name> |

## CREATING A DOCKER IMAGE



To create a docker image:

1. We create a Docker file (plain text file) – which will have command / configuration. The configuration defined what different program it contains and what it does when it starts as a container.
2. Once the docker file is created. It is supplied to docker client (docker cli) which in-turn provide the file to docker server. The docker server finally does the heavy lifting to create Docker image.

* To create our own custom image – we must identify the step and write the steps as command in Docker file.

### CREATING A DOCKER FILE

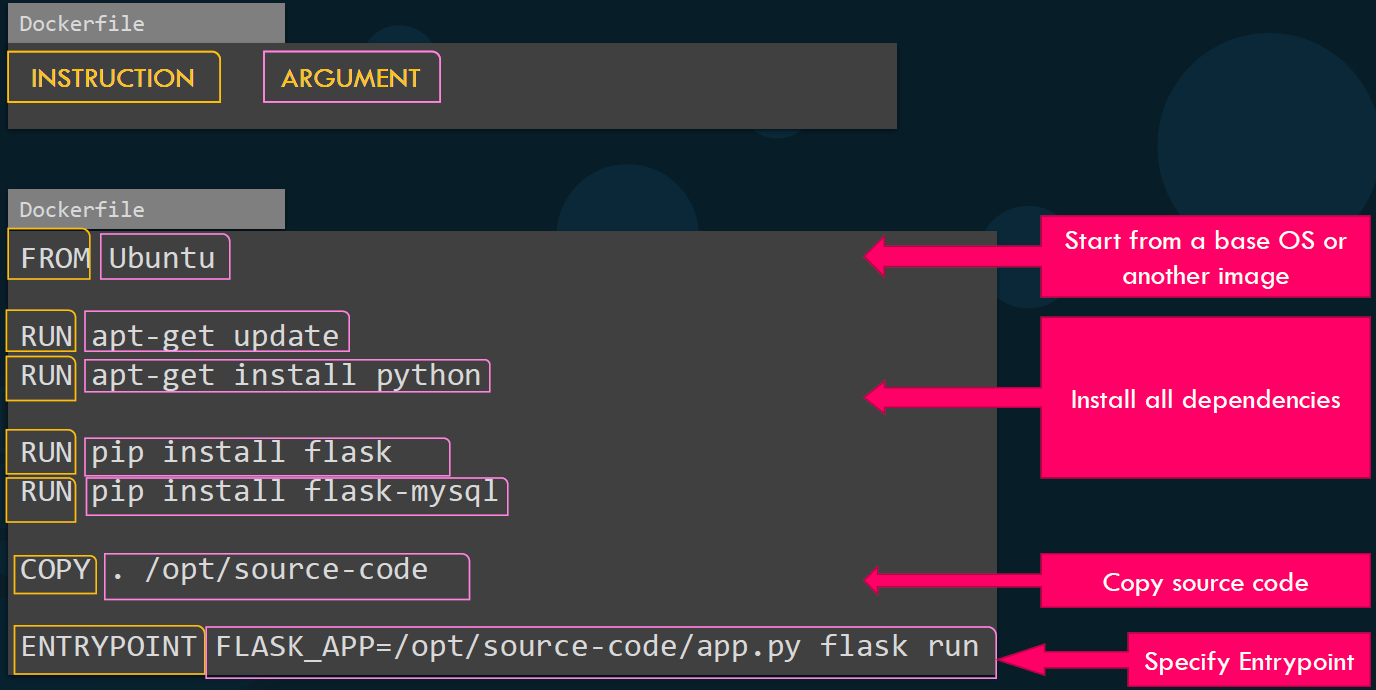
|  |  |
| --- | --- |
|  |  |

### DOCKERFILE IN DETAILS

|  |  |
| --- | --- |
|  | * Every line in a docker file is called instruction – which instruct to some basic preparation step on the custom  1. **FROM:** This is used to specify the docker image we want to use as a base. 2. **RUN:** Command while preparing the custom image 3. **CMD:** What should be executed when the image starts as a container. |

### DOCKER FILE FORMAT

* Docker files are written in instruction and argument format



### RUNNING THE DOCKER FILE

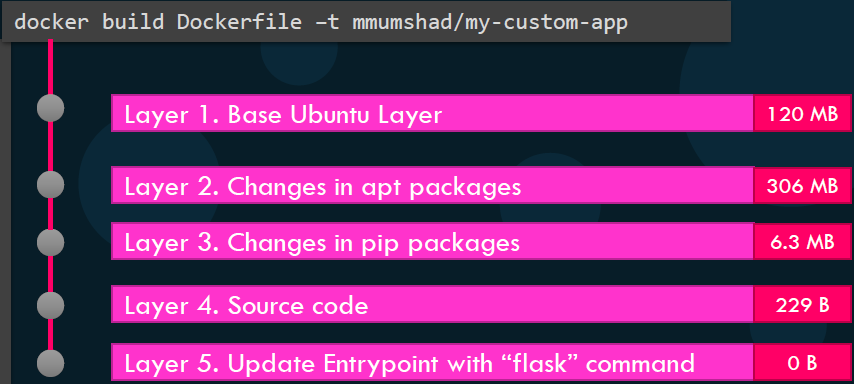
* Once the docker file is created – we can build the docker image

|  |  |
| --- | --- |
| TO BUILD THE DOCKER IMAGE | **docker build . -f Dockerfile -t <tag\_name>**  **docker build .** |
| TO RUN THE DOCKER IMAGE | **docker run <image\_name>** |
| TO PUSH THE DOCKER IMAGE TO DOCKER HUB | **docker push <account\_name>/<image\_Name>** |

|  |  |
| --- | --- |
| CREATE A DOCKER FILE |  |
| **DOCKER FILE** |  |
| RUNNING AND BUILDING A DOCKER IMAGE : **docker build <path of docker file>** | |

### LAYERED ARCHITECTURE OF DOCKER BUILD

1. When Docker builds the images, it builds these in a layered architecture.
2. Each line of instruction creates a new layer in the Docker image with just the changes from the previous



1. All the layers built are cached, so the layered architecture helps us restart Docker build from that step-in case it fails or if we were to add new steps in the build process, we wouldn't have to start all over again because all the layers built are cached by Docker.
2. This will help in rebuilding the image is faster as we don't have to wait for Docker to rebuild the entire image each time. This will be is helpful, especially when we update source code of the application as it may change more frequently.

Only the layer above the updated layers needs to be rebuilt.

### ENVIRONMENT VARIABLES IN DOCKER

Environment variables in Docker can be set and used in multiple ways.

1. **SETTING ENVIRONMENT VARIABLES IN DOCKERFILE**:  
   We can set environment variables directly in the Dockerfile using the ENV instruction.

**ENV ENV\_VAR\_NAME=value**

*Replace ENV\_VAR\_NAME with the name of the environment variable we want to set, and value with the desired value.*

1. **PASSING ENVIRONMENT VARIABLES AT BUILD TIME**:  
   We can pass environment variables to the docker build command using the **--build-arg** option.

**EXAMPLE : docker build --build-arg ENV\_VAR\_NAME=value .**

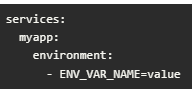
*In your Dockerfile, you can use the ARG instruction to define the build-time argument and then use the ENV instruction to set it as an environment variable.*

1. **SETTING ENVIRONMENT VARIABLES DURING CONTAINER RUNTIME:**  
   We can set environment variables when running a container using the **-e or --env option** with the **docker run command**.

**EXAMPLE : EXAMPL: docker run -e ENV\_VAR\_NAME=value <image-name>**

*Replace ENV\_VAR\_NAME with the name of the environment variable you want to set, value with the desired value, and <image-name> with the name of the Docker image.*

1. **USING ENVIRONMENT VARIABLES IN DOCKER COMPOSE**:  
   If you are using Docker Compose, you can define environment variables in the environment section of the service definition. Here's an example:



*Replace ENV\_VAR\_NAME with the name of the environment variable you want to set, and value with the desired value.*

NOTE : To inspect the environment variable on a running command make use of docker inspect

|  |  |
| --- | --- |
| docker inspect <container\_name / container\_id> |  |

#### EXAMPLE

Run a container named blue-app using image kodekloud/simple-webapp and set the environment variable APP\_COLOR to blue. Make the application available on port 38282 on the host. The application listens on port 8080.

**docker run -p 38282:8080 --name blue-app -e APP\_COLOR=blue -d kodekloud/simple-webapp**

* To know the env field from within a webapp container, run **docker exec -it blue-app env**

### COMMAND VERSUS ENTRYPOINT

* **Both CMD and ENTRYPOINT are instructions used to specify the command to run when a container is started from an image**.

#### CMD

* The CMD instruction specifies the default command and/or parameters for an executable in a container.
* It can be overridden by providing arguments when running the container.
* If multiple CMD instructions are specified in the Dockerfile, only the last one will take effect.
* The CMD instruction is typically used to provide a default command that can be easily overridden.

**CMD ["python", "app.py"]**

#### ENTRYPOINT

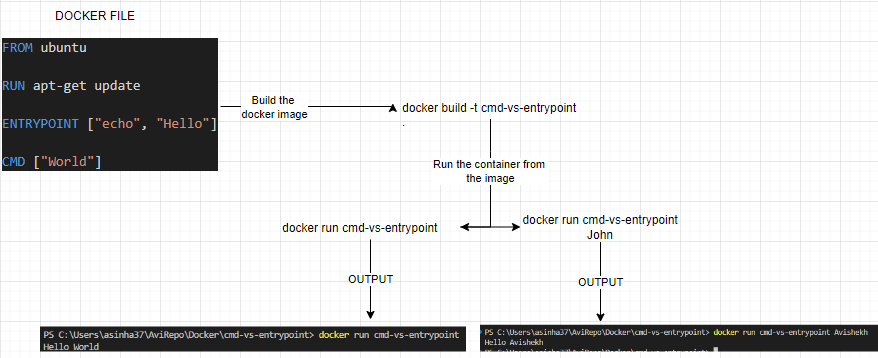
* **The ENTRYPOINT instruction specifies the command that will be executed when the container is started**.
* **ENTRYPOINT is the process that’s executed inside the container.**
* The ENTRYPOINT Dockerfile instruction sets the process that’s executed when the container starts.

**ENTRYPOINT ["/usr/bin/my-app"]**

* In this example, the container will run **/usr/bin/my-app**
* **CMD instruction sets the default arguments that are passed to the ENTRYPOINT process**. It determines the final form of the command string that will be executed.

|  |  |
| --- | --- |
| **ENTRYPOINT ["/usr/bin/my-app"]**  **CMD ["help"]** | The container will run : /**usr/bin/my-app help** |
| **ENTRYPOINT ["python"]**  **CMD ["app.py"]** | the container will run : **python app.py** |

* **CMD is the default set of arguments that are supplied to the ENTRYPOINT process. The CMD command can be overridden by providing arguments during container runtime.**
* When an image is created without an ENTRYPOINT, Docker defaults to using /bin/sh -c.
* **If multiple ENTRYPOINT instructions are specified, only the last one will be used.**



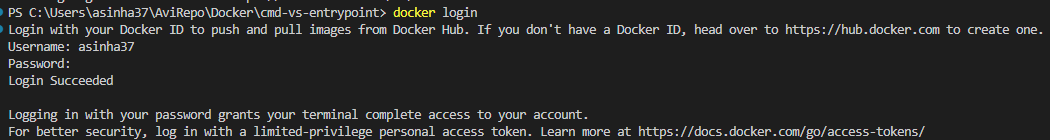
## CREATING AN IMAGE IN LOCAL MACHINE

* Application Repo: <https://github.com/mmumshad/simple-webapp-flask/tree/master>

## PUSHING DOCKER IMAGE

To push a Docker image to Docker Hub, follow these steps:

1. **LOG IN TO DOCKER HUB**: Use the docker login command to log in to Docker Hub from command line interface (CLI) and provide your Docker Hub username and password when prompted.



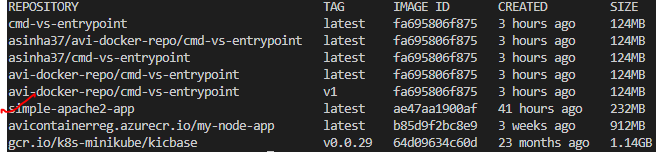
1. **TAG IMAGE**:
   1. Before pushing the image, ensure that it is properly tagged with your Docker Hub username and the repository name.
   2. Use the docker tag command to tag your image.

**docker tag <image-id> your-docker-username/your-repository-name**

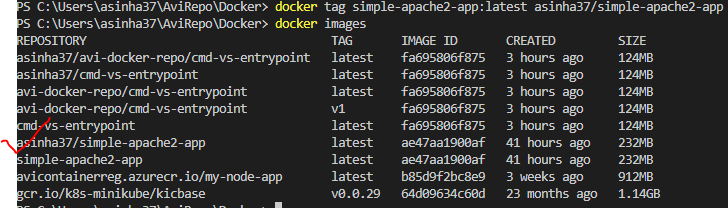
*Replace your-docker-username with your Docker Hub username and your-repository-name with the desired repository name*

**EXAMPLE**

1. In this example – We will push the “simple-apache2-app” image to docker hub



**COMMAND - CREATING TAG : docker tag simple-apache2-app:latest asinha37/simple-apache2-app**

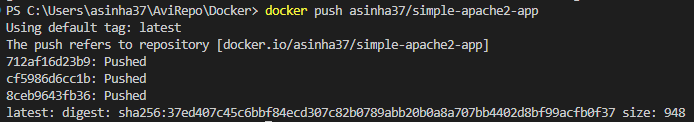


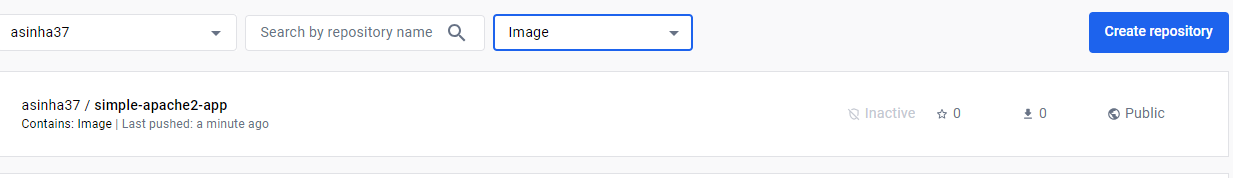
1. PUSH THE IMAGE: Use the docker push command to push the tagged image to Docker Hub.

**docker push your-docker-username/your-repository-name**

*Replace your-docker-username and your-repository-name with your Docker Hub username and repository name*

COMMAND : DOCKER PUSH - docker push asinha37/simple-apache2-app

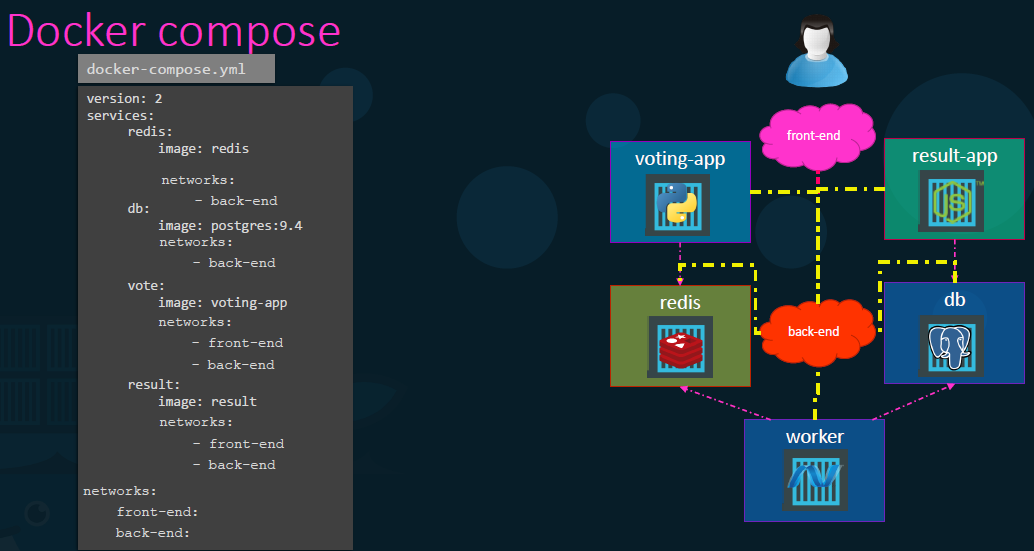




## DOCKER COMPOSE

* Docker Compose is a tool that allows us to define and manage multi-container Docker applications.
* It is used to simplify the process of running multiple interconnected containers as a single application stack.
* With Docker Compose, we can define a YAML file called **docker-compose.yml** that contains the configuration for the **application's services, networks, and volumes**.
* Each service defined in the Compose file represents a container that makes up the application.

### EXAMPLE



<https://docs.docker.com/compose/>

<https://docs.docker.com/engine/reference/commandline/compose/>

<https://github.com/dockersamples/example-voting-app>