Contents

[DOCKER 4](#_Toc154774249)

[WHAT IS DOCKER? 4](#_Toc154774250)

[WHY DO WE DOCKER 4](#_Toc154774251)

[WHAT ARE CONTAINERS 5](#_Toc154774252)

[SHARING KERNEL 5](#_Toc154774253)

[RUNNING LINUX CONTAINER ON WINDOWS 6](#_Toc154774254)

[CONTAINER VERSUS VM 6](#_Toc154774255)

[DOCKER ON WINDOWS 7](#_Toc154774256)

[DOCKER DESKTOP 7](#_Toc154774257)

[USING DOCKER CLIENT 8](#_Toc154774258)

[DOCKER IMAGES 9](#_Toc154774259)

[WHAT IS CONTAINER? 9](#_Toc154774260)

[RELATION BETWEEN CONTAINER AND IMAGE 11](#_Toc154774261)

[LINUX KERNEL 11](#_Toc154774262)

[DOCKER COMMANDS 12](#_Toc154774263)

[DOCKER RUN 12](#_Toc154774264)

[LIST ALL RUNNING CONTAINER 13](#_Toc154774265)

[STOPING THE RUNNING CONTAINER 13](#_Toc154774266)

[RESTARTING THE STOPPED CONTAINER 13](#_Toc154774267)

[REMOVING THE IMAGES AND CONTAINER 13](#_Toc154774268)

[INSPECT CONTAINER 14](#_Toc154774269)

[DOCKER IMAGES COMMANDS 14](#_Toc154774270)

[DOCKER PRUNE 14](#_Toc154774271)

[REMOVING STOPPED CONTAINER AUTOMATICALLY 14](#_Toc154774272)

[EXECUTING A TASK IN A RUNNING CONTAINER 14](#_Toc154774273)

[EXAMPLES 15](#_Toc154774274)

[ATTACHED AND DETACHED MODE OF RUNNING CONTAINER 15](#_Toc154774275)

[ATTACHED MODE 15](#_Toc154774276)

[DETACHED MODE 15](#_Toc154774277)

[TAGGING 15](#_Toc154774278)

[CONVENTION TO TAG IMAGES 15](#_Toc154774279)

[INTERACTIVE MODE - EXECUTING COMMAND IN A RUNNING CONTAINER 16](#_Toc154774280)

[EXAMPLE 17](#_Toc154774281)

[GETTING TERMINAL ACCESS OF CONTAINER 17](#_Toc154774282)

[CONTAINER - PORT MAPPING 18](#_Toc154774283)

[STEPS TO MAP THE PORT 18](#_Toc154774284)

[EXAMPLE 19](#_Toc154774285)

[DATA PERSISTENCE IN DOCKER CONTAINER 19](#_Toc154774286)

[CONTAINER LOGS 19](#_Toc154774287)

[CREATING A DOCKER IMAGE 20](#_Toc154774288)

[CREATING A DOCKER FILE 20](#_Toc154774289)

[DOCKERFILE IN DETAILS 20](#_Toc154774290)

[DOCKER FILE FORMAT 20](#_Toc154774291)

[DOCKER FILE COMMANDS 21](#_Toc154774292)

[RUNNING THE DOCKER FILE 22](#_Toc154774293)

[LAYERED ARCHITECTURE OF DOCKER BUILD 23](#_Toc154774294)

[ANGuments AND ENVIRONMENT VARIABLES IN DOCKER 23](#_Toc154774295)

[**ENVironment VARIABLES – RUNTIME** 24](#_Toc154774296)

[ARGUMENTS – BUILD TIME ARGUMENTS 25](#_Toc154774297)

[COMMAND VERSUS ENTRYPOINT 26](#_Toc154774298)

[CMD 26](#_Toc154774299)

[ENTRYPOINT 26](#_Toc154774300)

[DOCKER STORAGE 26](#_Toc154774301)

[DOCKERS LAYERED ARCHITECTURE 27](#_Toc154774302)

[COPY ON WRITE 29](#_Toc154774303)

[DOCKER VOLUME AND BIND MOUNTS 30](#_Toc154774304)

[VOLUME 30](#_Toc154774305)

[BIND MOUNTS 31](#_Toc154774306)

[DOCKER IGNORE FILE 33](#_Toc154774307)

[EXAMPLE -1 34](#_Toc154774308)

[EXAMPLE -2 34](#_Toc154774309)

[DOCKER NETWORKING 34](#_Toc154774310)

[COMMUNICATION BETWEEN CONTAINER AND HOST 34](#_Toc154774311)

[COMMUNICATION BETWEEN CONTAINERS 34](#_Toc154774312)

[DOCKER NETWORK DRIVERS 36](#_Toc154774313)

[EMBEDDED DNS 38](#_Toc154774314)

[ASSOCIATING A CONTAINER TO A NETWORK 39](#_Toc154774315)

[DOCKER REGISTRY 39](#_Toc154774316)

[IMAGE NAMES IN REGISTRY 39](#_Toc154774317)

[RUNNING A PRIVATE DOCKER REPOSITORY 39](#_Toc154774318)

[DEPLOYING PRIVATE REGISTRY IN ORGANIZATION 40](#_Toc154774319)

[PUSHING DOCKER IMAGE 40](#_Toc154774320)

[LINKING THE CONTAINER – LEGACY METHOD 41](#_Toc154774321)

[DOCKER COMPOSE - MULTI-CONTAINER ORCHESTRATION 42](#_Toc154774322)

[WRITING THE DOCKER COMPOSE FILE 42](#_Toc154774323)

[DOCKER COMPOSE PROJECT 44](#_Toc154774324)

[STEPS TO SET UP DOCKER COMPOSE 46](#_Toc154774325)

[MAINTANING CONTAINER USING DOCKER COMPOSE 47](#_Toc154774326)

[DOCKER ENGINE 48](#_Toc154774327)

[COMPONENTS OF DOCKER ENGINE 48](#_Toc154774328)

[CONTAINER ORCHESTRATION 50](#_Toc154774329)

[DOCKER SWARN 50](#_Toc154774330)

[A DOCKER PROJECT 51](#_Toc154774331)

[STEPS TO CREATE THE PROJECT 51](#_Toc154774332)

[SETTING UP BASE IMAGE 52](#_Toc154774333)

[PORT MAPPING 52](#_Toc154774334)

[WORKDIR 53](#_Toc154774335)

[MINIMIZING THE CACHE BURSTING AND REBUILDS 53](#_Toc154774336)

[DOCKER COMPOSE- VOTING APP 54](#_Toc154774337)

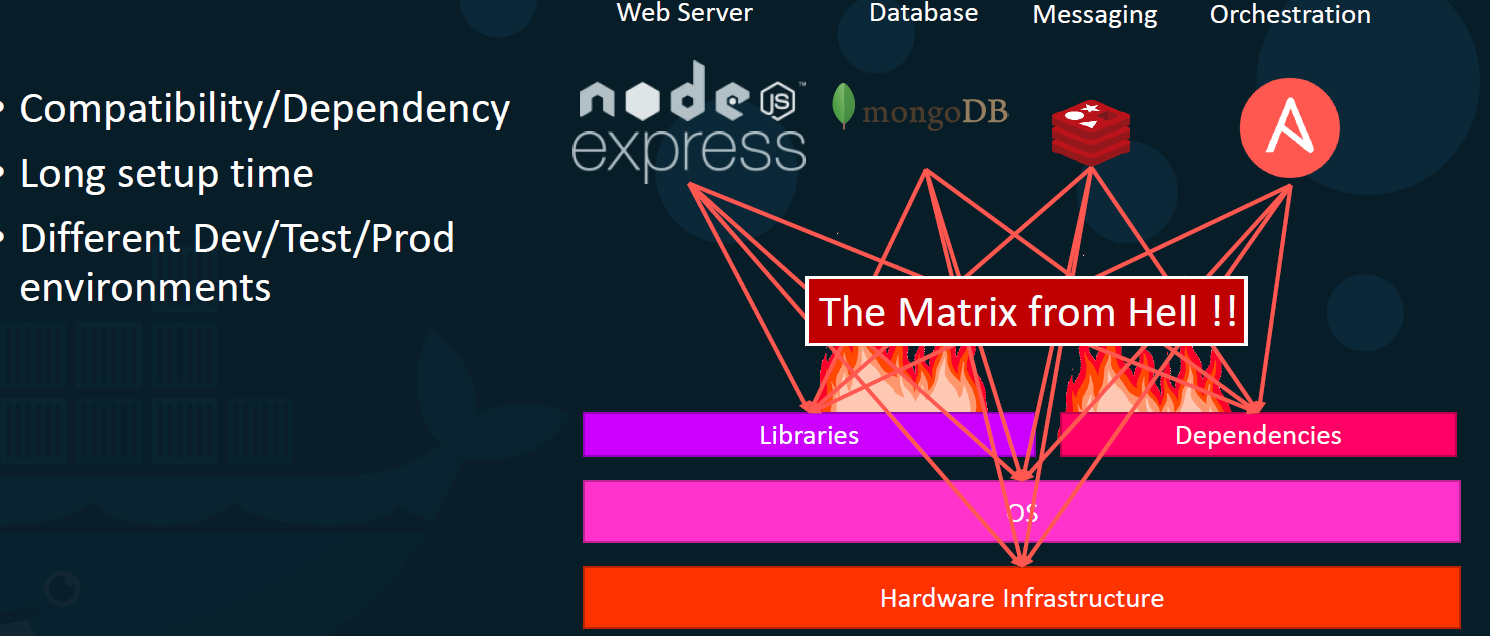
# DOCKER

## WHAT IS DOCKER?

### WHY DO WE DOCKER

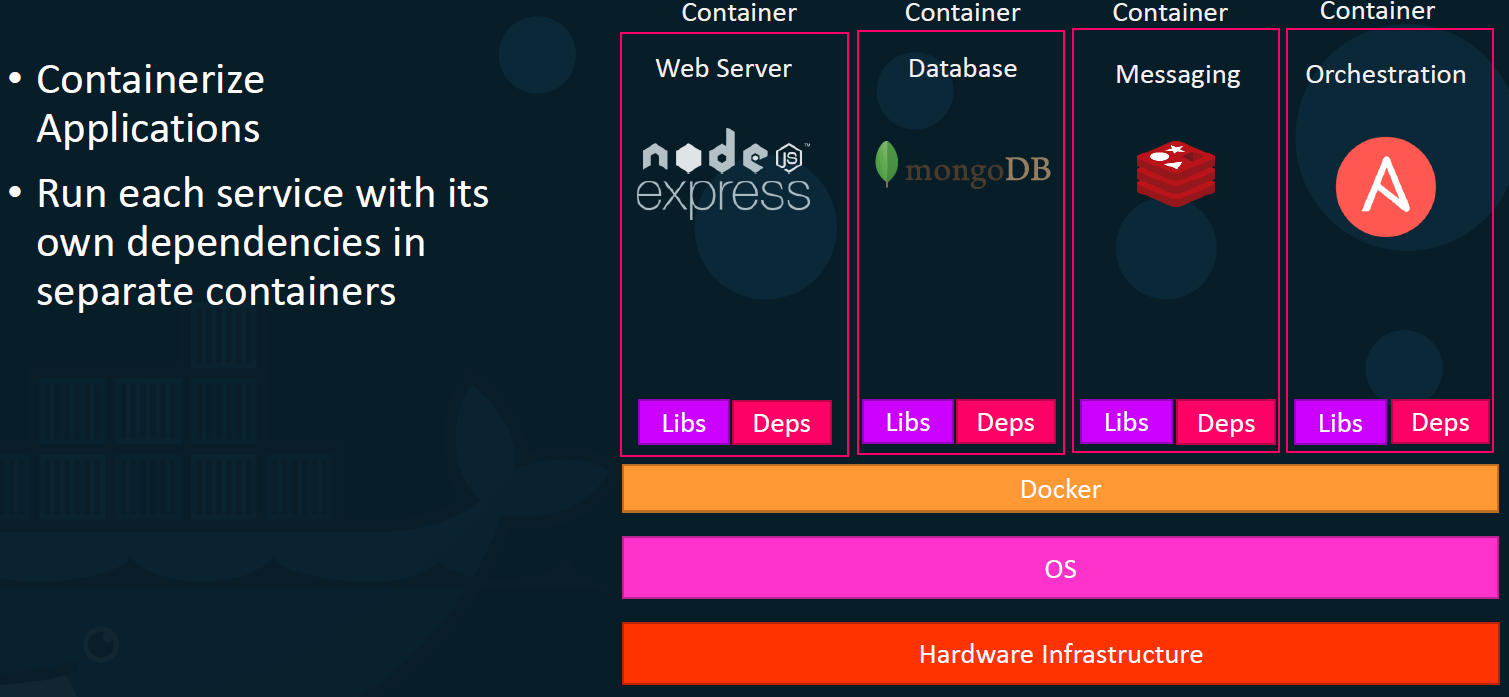
USE CASE

* Let’s consider we have to set up an application stack including various different technologies like
  + Web server using NodeJS
  + Database such as MongoDB
  + Messaging system like Redis
  + An orchestration tool like Ansible.
* We will have lot of issues developing this application stack with all these different components. The issues can be.



* **Compatibility with the underlying OS** - We have to ensure that all these different services were compatible with the version of OS.
* **Compatibility between these services and the libraries and dependencies on the OS** - For example - We've had issues where one service requires one version of a dependent library whereas another service requires another version.
* If the architecture of the application changes over time - We've had to upgrade to newer versions of these components or change the database etc. And every time something changed we had to go through the same process of checking compatibility between these various components and the underlying infrastructure.

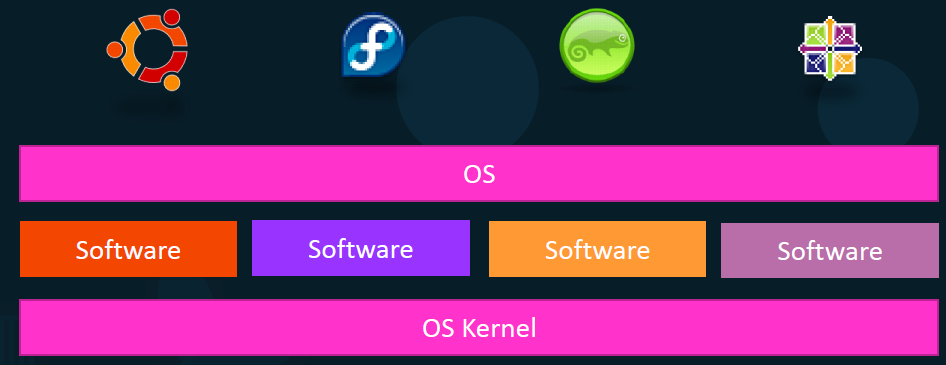
**THIS COMPATIBILITY METRICS ISSUE IS USUALLY REFERRED TO AS THE MATRIX FROM HELL NEXT.**



* **With docker we can be able to run each component in a separate container with its own dependencies and its own libraries - all on the same VM and the OS but within separate environments or containers we just had to build the docker configuration once and all our developers could now get started with a simple Docker run command irrespective of what the underlying operating system they run.**

### WHAT ARE CONTAINERS

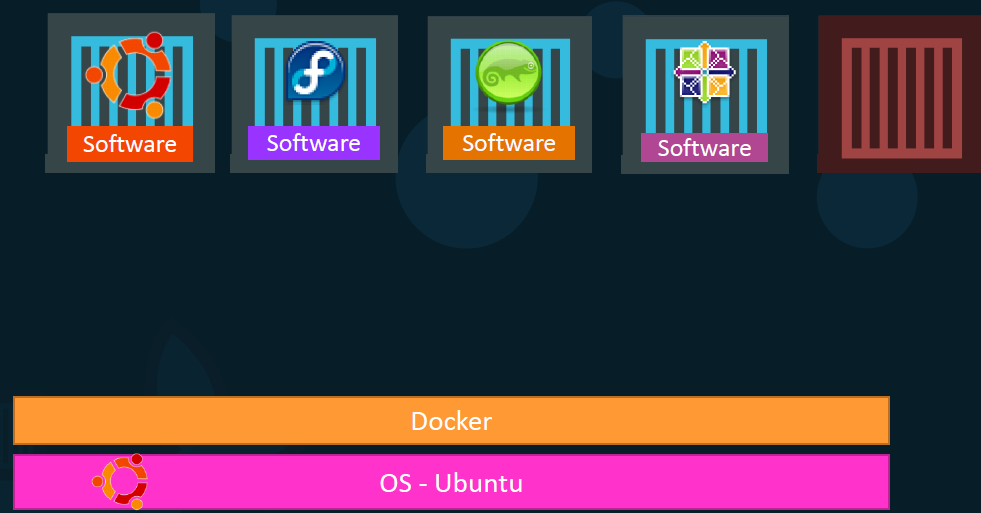
* Containers are completely isolated environments.
* They can have their own processes or services their own network interfaces their own mounts just like virtual machines except they all share the same OS kernel.



BASIC CONCEPTS OF OPERATING SYSTEM

* Operating systems like Ubuntu, Fedora, Suse or Centos – they all consist of two things
  + **AN OS KERNEL AND**
  + **SET OF SOFTWARES.**
* The OS kernel is responsible for interacting with the underlying hardware while the OS kernel remains the same which is Linux.
* ***Hence - In this case it's the software above it that makes these operating systems different.*** This software may consist of a different user interface drivers compilers file managers developer tool etc. So, we have a common Linux kernel shared across all OSes and some custom software that differentiate operating systems from each other.

### SHARING KERNEL



* The docker containers share the underlying kernel. Let's say we have a system with an Ubuntu OS with Docker installed on it Docker can run any flavour of OS on top of it as long as they are all based on the same kernel, In above case its Linux.
* For example - If the underlying OS is Ubuntu - Docker can run a container based on another distribution like Debian

fedora, suse or centos.

### RUNNING LINUX CONTAINER ON WINDOWS

#### 2WSL - WINDOWS SUBSYSTEM FOR LINUX

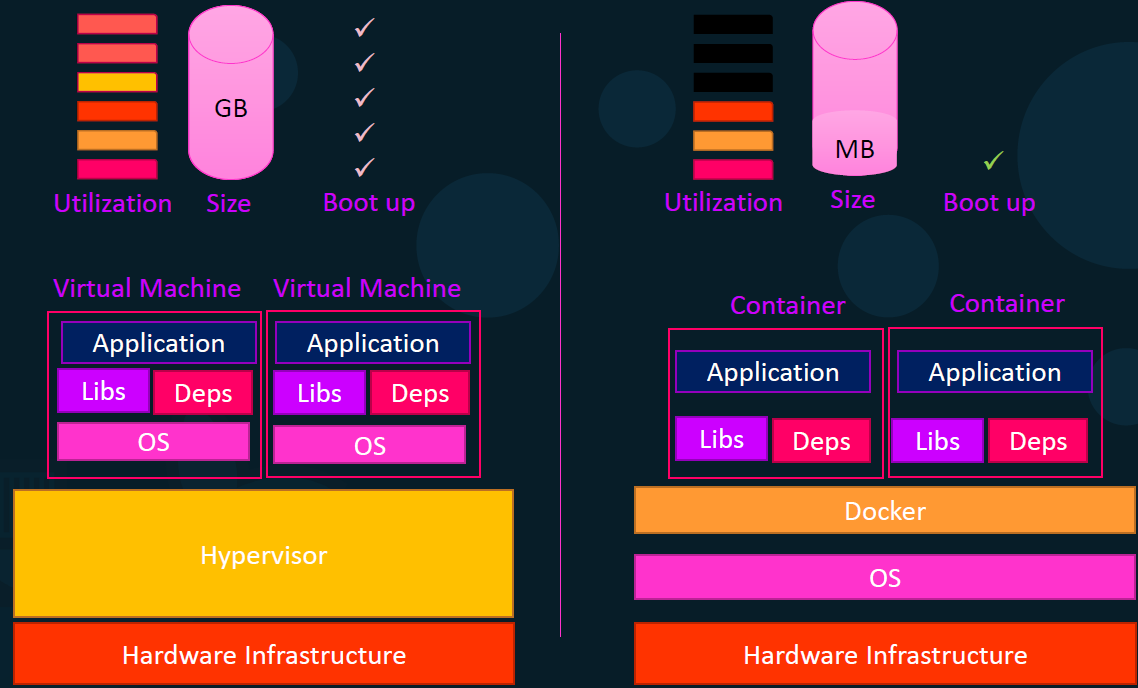
**WSL (Windows Subsystem for Linux) in Docker provides the ability to run Linux containers on Windows machines. It is a compatibility layer that enables developers to use Linux tools and workflows natively on Windows.**  
  
THE PURPOSE OF WSL IN DOCKER IS TO ADDRESS THE FOLLOWING SCENARIOS:

* DEVELOPMENT ENVIRONMENT CONSISTENCY:
  + With WSL, developers can use the same tools and workflows on both Windows and Linux systems.
  + This allows for consistent development environments across different platforms and simplifies collaboration between developers working on different operating systems.
* SEAMLESS INTEGRATION:
  + WSL allows running Linux-based Docker containers on a Windows machine without the need for a separate Linux virtual machine.
  + It provides a lightweight and efficient way to run Linux workloads alongside Windows applications, enabling developers to leverage the benefits of both operating systems.
* IMPROVED PERFORMANCE:
  + WSL provides a faster and more efficient runtime for Linux containers on Windows, compared to running a full-fledged Linux VM.
  + It achieves this by utilizing the Windows kernel and translating Linux system calls to Windows equivalents, reducing the overhead of virtualization.
* ACCESS TO LINUX TOOLS AND LIBRARIES:
  + WSL provides access to a wide range of Linux tools, utilities, and libraries, allowing developers to use their preferred Linux-based toolchain and ecosystem.
  + This can be beneficial for developers working with technologies that are primarily Linux-based, such as scripting languages, package managers, and build systems.

*Overall, WSL in Docker enhances the development experience for developers using Windows machines by bridging the gap between Windows and Linux environments, enabling seamless integration and consistent development workflows.*

|  |
| --- |
| NOTE: Unlike hypervisor is Docker is not meant to virtualize and run different operating systems and kernels on the same hardware. The main purpose of Docker is to package and containerized applications and to ship them and to run them anywhere |

### CONTAINER VERSUS VM



* In case of Docker we have the underlying hardware infrastructure and then the OS and then Docker installed

on the OS. Docker then manages the containers that run with libraries and dependencies.

* In case of virtual machines- We have the hypervisor(like EXS) on the hardware and then the virtual machines on them, each virtual machine has its own OS inside it 🡪 Then the dependencies and- 🡪 then the application .
* *This overhead causes higher utilization of underlying resources as there are multiple virtual operating systems and kernel running the virtual machines also consume higher this space as each VM is heavy and is usually in gigabytes in size whereas docker containers are lightweight and are usually in megabytes in size.*
* This allows Docker containers to boot up faster usually in a matter of seconds whereas VMs as we know takes minutes to boot up as it needs to boot up the entire operating system.
* Note: Docker has less isolation as more resources are shared between the containers like the kernel whereas VMS have complete isolation from each other since VMS don't rely on the underlying OS or kernel.

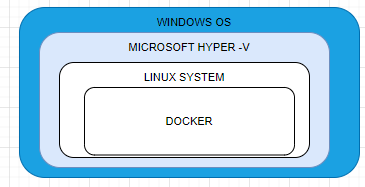
## DOCKER ON WINDOWS

OPTIONS AVAILABLE FOR DOCKER ON WINDOWS.

1. **USING DOCKER TOOLBOX**
2. **DOCKER DESKTOP FOR WINDOWS.**

### DOCKER DESKTOP

RUNNING THE LINUX CONTAINER FOR WINDOW OS



***Note - Microsoft has its native virtualization technology called Hyper-V***

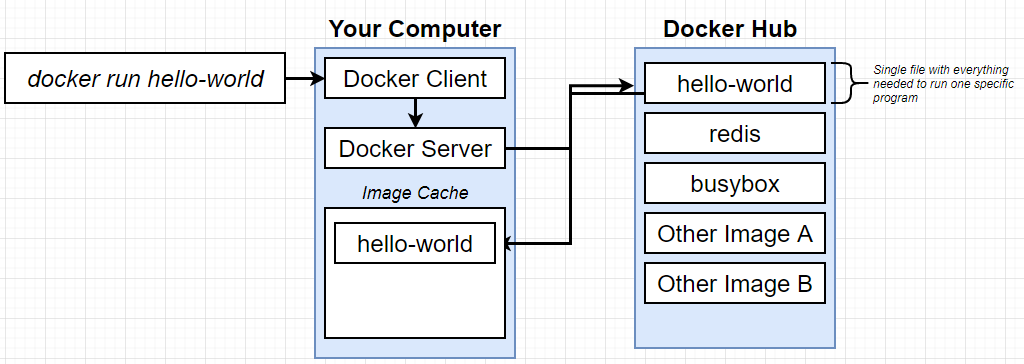
* For Windows OS. We make use of Hyper-V 🡪 Linux System 🡪 Docker

*Note: This option is only supported for Windows 10 enterprise or professional edition and on Windows Server 2016 because both these operating systems come with hyper v support by default.*

|  |  |
| --- | --- |
|  | DOCKER DESKTOP   * **Microsoft announced support for Windows containers - we can now package applications windows applications into Windows docker containers and run them on the windows docker host using Docker desktop for Windows** * The default option is to work with Linux containers but if we want to run Windows containers, then we must explicitly configure **Docker for Windows to switch to using Windows containers** |

## 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 |  |

## DOCKER IMAGES

* In Docker, a container image is built from a series of read-only layers.
* Each layer represents a specific instruction or change made in the Dockerfile during the image build process.
* These layers are stacked on top of each other, and the final layer is the topmost layer, which represents the current state of the container image.

KEY POINTS ABOUT DOCKER IMAGE LAYERS:

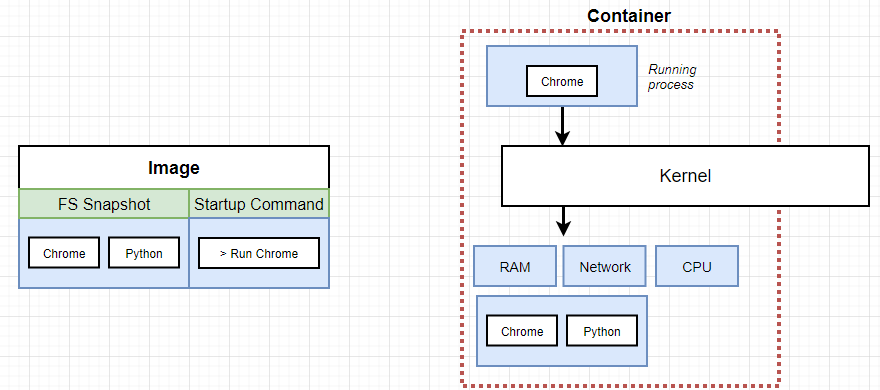
1. LAYERED FILE SYSTEM:
   1. Docker uses a layered file system, typically Union File System (UFS), to construct and manage these layers efficiently.
   2. Each layer is essentially a set of differences made on top of the previous layer, reducing the overall image size and enabling faster image building and distribution.
2. REUSABILITY:
   1. Docker image layers are designed to be reusable.
   2. If multiple images share the same base layers, Docker can reuse those layers during the build process, saving disk space and reducing the time required to pull or build images.
3. CACHING:
   1. Docker uses layer caching to speed up image builds. When a Dockerfile instruction is executed, Docker checks if the layer already exists in the local cache. If it does, Docker reuses the cached layer instead of rebuilding it, unless the instruction modifies the layer.
4. LAYER INDEPENDENCE:
   1. Each layer is independent and immutable.
   2. When an image is modified or a container is started, Docker only needs to create or modify the topmost layer, keeping the lower layers intact. This approach provides better performance and allows for easier image updates.
5. LAYERED UPDATES:
   1. When we update an image by adding new instructions to the Dockerfile and rebuild it, only the layers affected by the changes are rebuilt.
   2. Docker reuses the existing layers that remain unchanged, which makes image updates quicker and more efficient.

## 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

|  |  |
| --- | --- |
| For docker command help | **docker --help** |
| Help for specific command. For example, to get help for “ps -a” - command | **docker ps -a --help** |

### DOCKER RUN

|  |  |
| --- | --- |
| * Docker run command creates and run a container from an image. * By default the **docker run** command runs the container in a attached mode but the “**docker start**” (this command restart the exited/ stopped container) run the container , by default in in detached mode. |  |
| * TO RUN CONTAINER using docker run command in the detached mode . We have to explictly start the container in detached mode | **docker run -d <image\_name>/<image\_id>** |

* If we started a container in detached mode (i.e. with -d), we can still attach to it afterwards without restarting the Container

COMMAND: **docker attach CONTAINER**

This attaches to a running Container with an ID or name of CONTAINER.

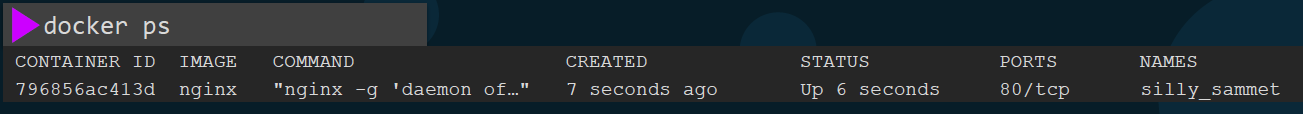
#### OVERRIDING DEFAULT COMMANDS

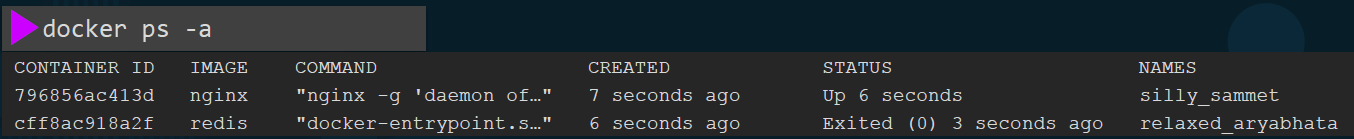
* “docker run” command by default create+ start the container. We can override the default behavior of these commands. e.g.

|  |  |
| --- | --- |
| ***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. |

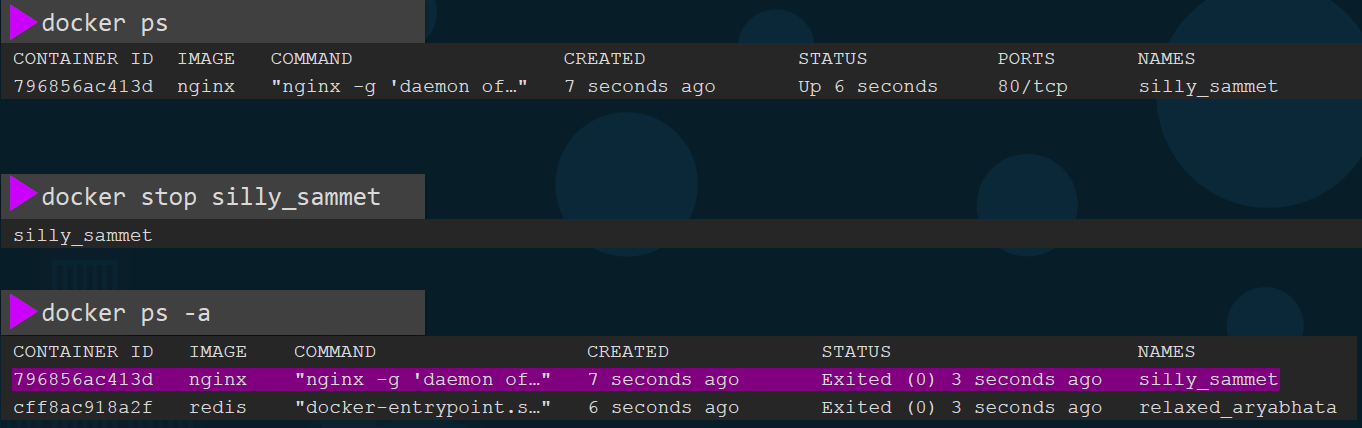




### STOPING THE RUNNING CONTAINER

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

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

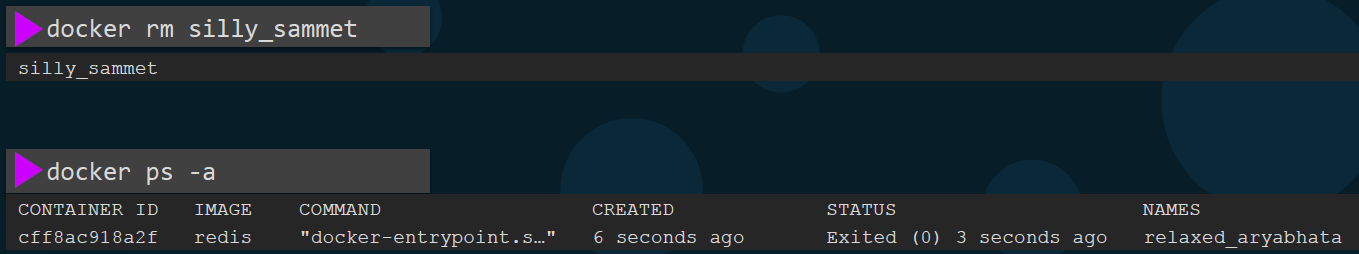


### RESTARTING THE STOPPED CONTAINER

|  |  |
| --- | --- |
| RESTARTING THE STOPPED CONTAINER | docker start <image\_id>/ <image\_name>   * The “**docker start**” (this command restart the exited/ stopped container) run the container , by default in in detached mode. |

### REMOVING THE IMAGES AND CONTAINER

REMOVES THE RUNNING CONTAINER

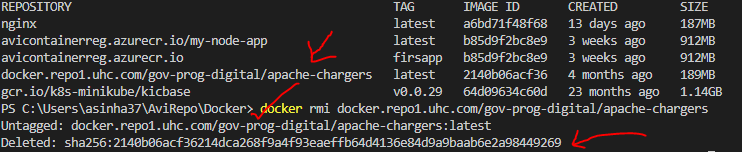


### INSPECT CONTAINER

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

### DOCKER IMAGES COMMANDS

|  |  |
| --- | --- |
| 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. |

## DOCKER PRUNE

* The docker prune command is used to clean up unused Docker resources, such as containers, images, networks, and volumes. It helps to free up disk space and remove unnecessary artifacts from your Docker environment.

**THERE ARE DIFFERENT TYPES OF PRUNING OPTIONS AVAILABLE IN DOCKER:**

|  |  |
| --- | --- |
| CONTAINER PRUNE: To remove stopped containers | docker container prune |
| IMAGE PRUNE: To remove unused images | docker image prune |
| NETWORK PRUNE: To remove unused networks | docker network prune |
| VOLUME PRUNE: To remove unused volumes | docker volume prune |

* To execute all the pruning operations at once, we can use the command docker system prune or docker system prune -a (including all unused images). However, be cautious as this command will remove all unused resources, and the operation cannot be undone.

## REMOVING STOPPED CONTAINER AUTOMATICALLY

|  |  |
| --- | --- |
| REMOVE THE CONTAINER WHEN IT IS STOPPED | **docker run -p 3000:8000 -d –rm <image\_id>/<image\_name>** |

## 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 |

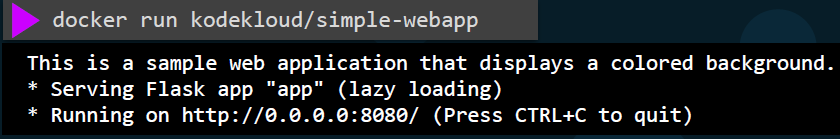
### EXAMPLES

|  |  |
| --- | --- |
| **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

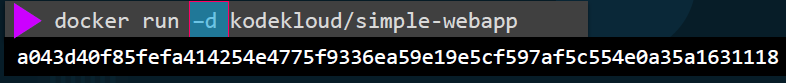
### ATTACHED MODE

* 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.

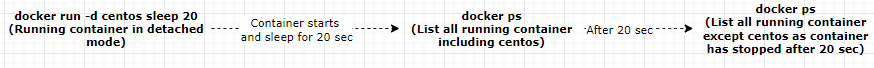


* To attach back to the running container later run the docker attach command and specify the name or I.D. of the docker container.

**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.

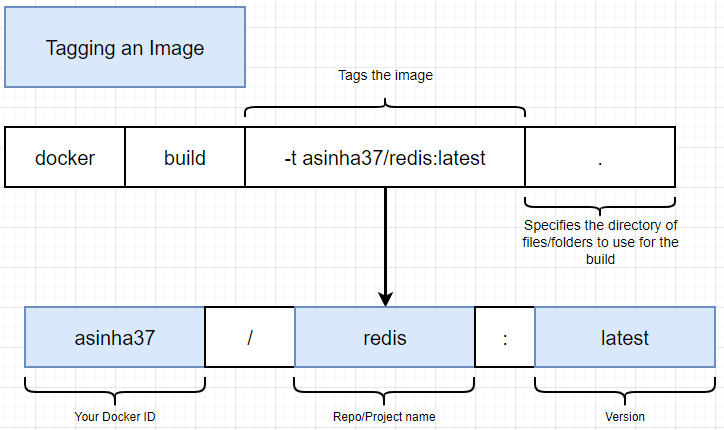


## TAGGING

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

### CONVENTION TO TAG IMAGES

* The convention to tag the images : **<Docker ID>/<Repo> or <Project Name>: version .** The version can be a number or “latest”.
  + TO BUILD THE IMAGE WITH A TAG: **docker build -t asinha37/redis .**
  + TO RUN THE IMAGE: **docker run asinha37/redis**

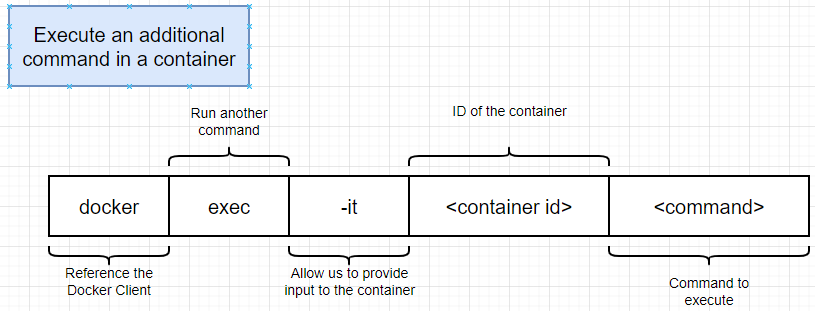


|  |  |
| --- | --- |
| **Pulling an image with a specific tag** | **docker pull <image\_name>:<tag>** |
| **Replace <image\_name> with the name of the image, and <tag> with the desired version or variant of the image** | Example- to pull the "nginx" image with the "1.21.1" tag  **docker pull nginx:1.21.1** |
| **Tagging an existing image (Retag)** | **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** |

## INTERACTIVE MODE - EXECUTING COMMAND IN A RUNNING CONTAINER

Let’s consider an example of redis server.

|  |  |
| --- | --- |
|  | * To interact with the redis server we make use of “redis-cli”. This can be easy when we have redis-server installed in a local machine. We can start the “redis server ” from one terminal and then start the redis-cli in another terminal. * When the redis-server is running as docker container – then redis-cli can only interact if it is running in “container’s” terminal itself.   **HENCE, WE NEED A WAY TO RUN COMMAND IN A RUNNING CONTAINER (AS SHOWN IN DIAGRAM)** |

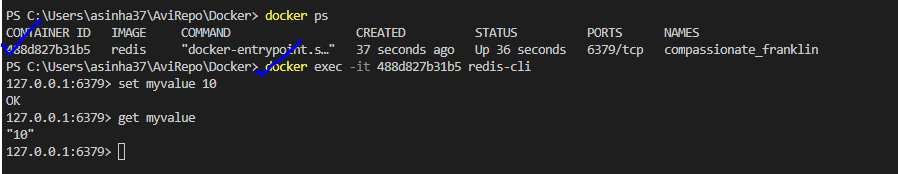


* We can execute commands within the running container using above docker command

**EXAMPLE : To execute “redis-cli” command in running “redis-server” container**

**COMMAND : docker exec -it <container\_id> redis-cli**

**RUNNING redis-cli in running redis container**



* The **-it is combination of two flags**
* -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 in interactive mode - 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 |

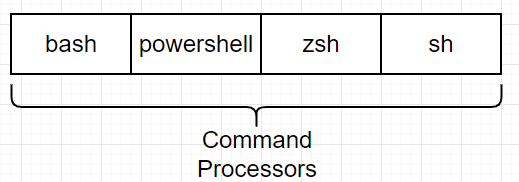
### GETTING TERMINAL ACCESS OF CONTAINER

|  |  |
| --- | --- |
| COMMAND TO ACCESS TERMINAL OF THE CONTAINER | docker exec -it 488d827b31b5 **sh** |
| * NOTE : “sh” is a command processor * After getting the terminal access we can able to execute all the linux command in the running container’s terminal | |

**COMMAND PROCESSORS**

* Command processors, also known as command-line interpreters or command-line shells, are software programs that interpret and execute commands entered by users through a command-line interface (CLI).
* They provide a text-based interface for users to interact with the operating system or other software applications by entering commands.
* Command processors typically read user input, parse the commands, and execute the corresponding actions.

**EXAMPLES OF COMMAND PROCESSORS**



## CONTAINER - PORT MAPPING

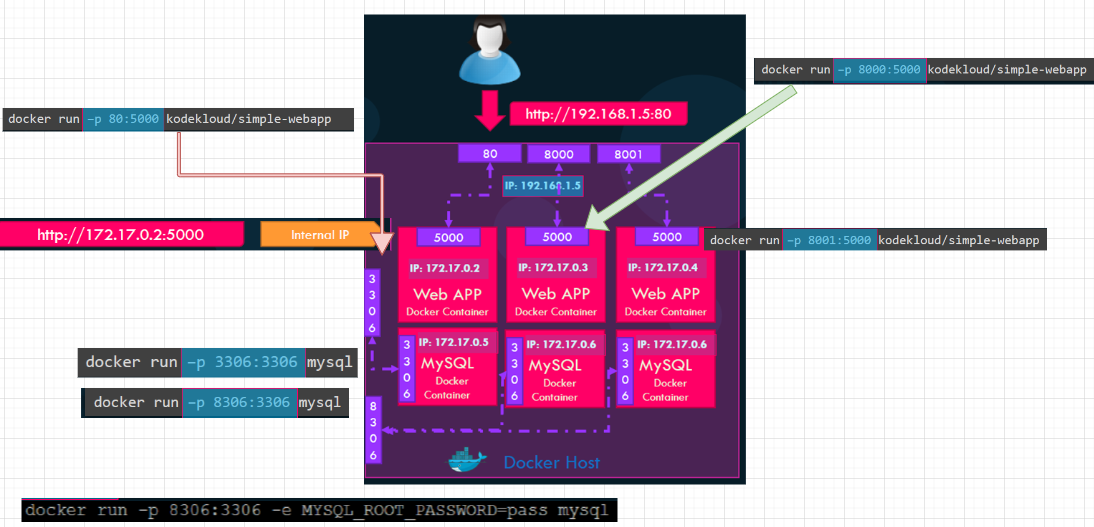
* Port mapping in Docker allows us to map network ports between the host machine and the container.
* This enables us to access services running inside a container from the host machine or other machines on the network.

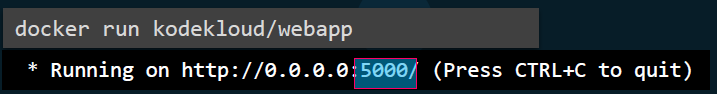
|  |  |
| --- | --- |
| * Use the -p (or --publish) flag with the docker run command * <host\_port>  : The port on the host machine that we want to map. * <container\_port> : The port on the container that we want to expose. | **SYNTAX**  **docker run -p <host\_port>:<container\_port> <image\_name>** |
| 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. |

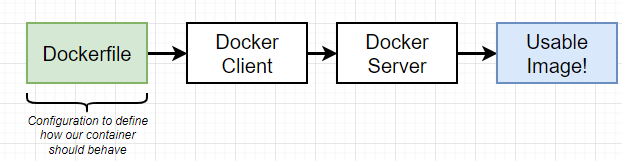
* Volumes can be managed using the docker volume command or specified in a Docker Compose file.

**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

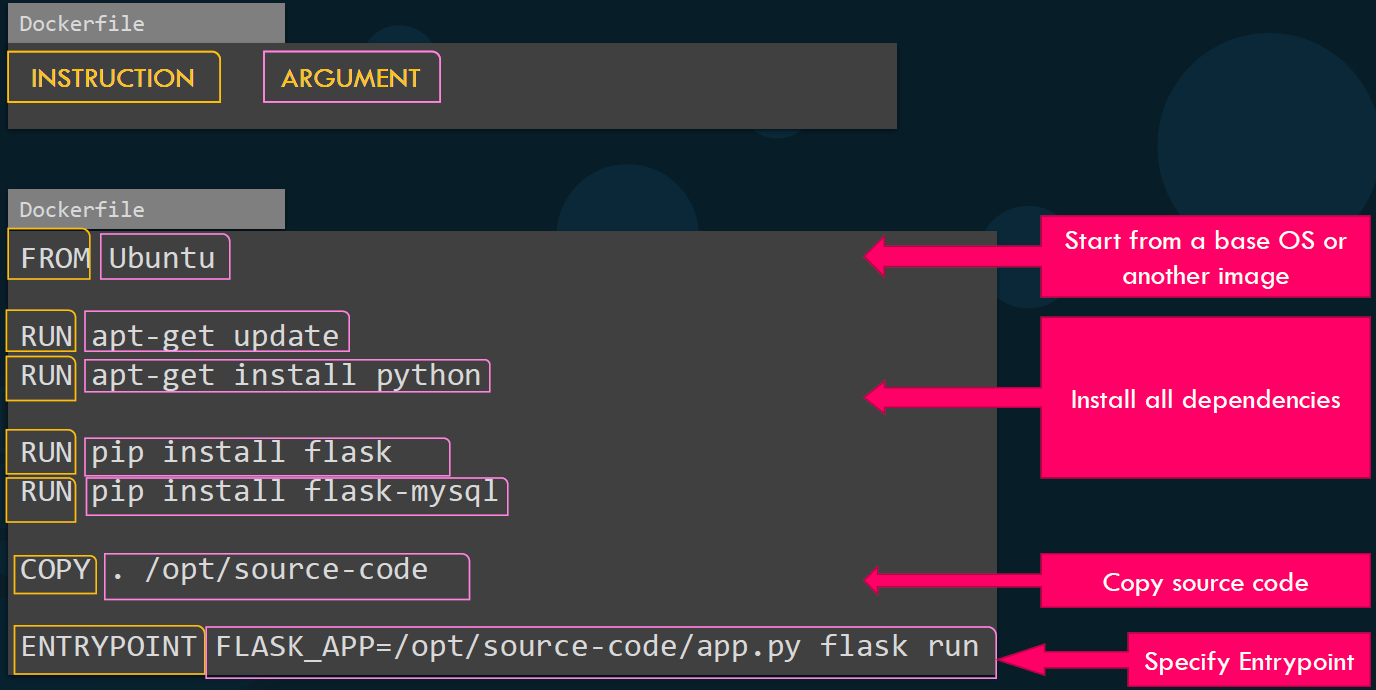
|  |  |
| --- | --- |
|  |  |

### 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



### DOCKER FILE COMMANDS

* **FROM**:
  + This instruction sets the base image for the subsequent instructions in the Dockerfile.
  + It is typically the first instruction in a Dockerfile and specifies the starting point for your image.
  + For example, FROM node:14 sets the Node.js version 14 image as the base.
* **ENTRYPOINT**:
  + This instruction specifies the default command that will be executed when a container is started from the image.
  + It allows us to define the main executable for the container.
  + For example, ENTRYPOINT ["npm", "start"] sets the npm start command as the entry point.
* **CMD**:
  + This instruction provides default arguments to the ENTRYPOINT or allows us to specify a command to be executed when the container is run.
  + It can be overridden by passing arguments when running the container.
  + For example, CMD ["app.js"] sets app.js as the default command to be executed.

|  |
| --- |
| **OVERIDING THE CMD COMMAND**   * When running a container, we can override the CMD instruction by passing arguments as command-line arguments.   **EXAMPLE**: Let's say Dockerfile has the following CMD instruction: **CMD ["npm", "start"]**   * By default, this command will be executed when the container starts, running the npm start script. However, we can override it by providing additional arguments when running the container. * For instance, if you want to run a different script, such as npm run dev, we can pass it as an argument when running the container:   **docker run my-app npm run dev**   * In this example, my-app is the name of the Docker image, and npm run dev is passed as an argument. This will override the default npm start command defined in the CMD instruction of the Dockerfile. |

* **COPY**:
  + This instruction copies files or directories from the build context (the directory where the Dockerfile resides) into the image. It takes two arguments: the source path in the build context and the destination path in the image. For example, COPY ./app /usr/src/app copies the app directory from the build context to /usr/src/app in the image.
* EXPOSE:
  + This instruction documents the ports that the container will listen on at runtime.
  + It does not actually publish the ports.
  + It's typically used for informational purposes to indicate which ports are intended to be exposed.
  + For example, EXPOSE 8080 documents that the container listens on port 8080.
* USER:
  + This instruction sets the user or UID that the container process will run as when the container is started.
  + It is useful for running the container with a non-root user for security reasons.
  + For example, USER node sets the user to "node".
* WORKDIR:
  + This instruction sets the working directory for any subsequent instructions in the Dockerfile.
  + It allows us to specify a directory path where subsequent instructions will be executed.
  + For example, WORKDIR /app sets the working directory to /app.
* **SHELL**:
  + This instruction sets the default shell used for the shell form of the CMD, RUN, and ENTRYPOINT instructions.
  + It allows you to customize the shell used to run commands in the Dockerfile.
  + For example, SHELL ["/bin/bash", "-c"] sets the shell to Bash.
* RUN:
  + This instruction executes commands in a new layer on top of the current image and commits the changes.
  + It is used to install packages, run build commands, or perform any other necessary setup steps in the image.
  + For example, RUN npm install installs the Node.js dependencies specified in the package.json file.

### RUNNING THE DOCKER FILE

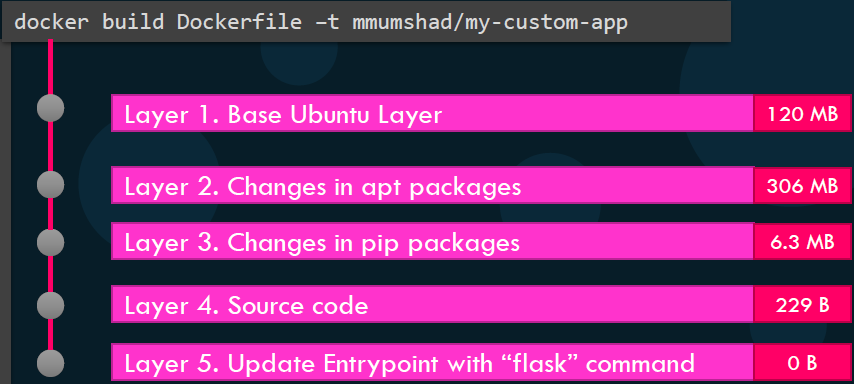
* Once the docker file is created – we can build the docker image

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

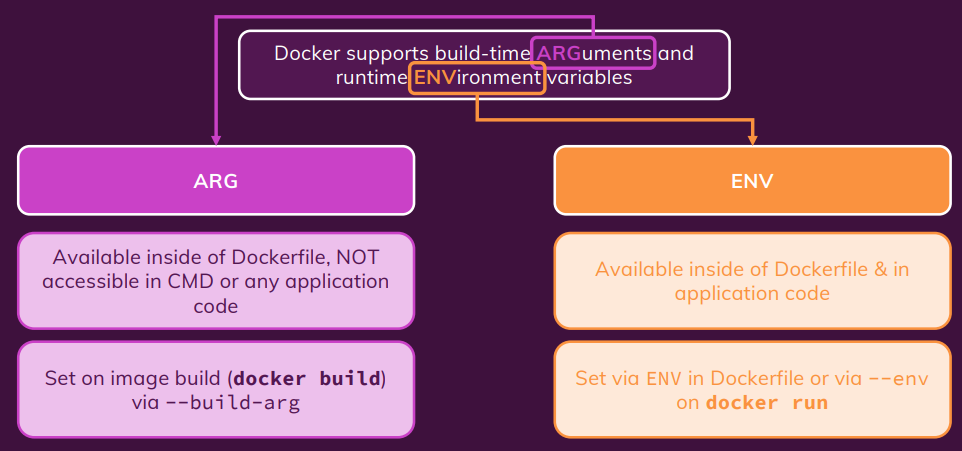
1. When Docker builds the images, it builds these in a layered architecture.
2. Each line of instruction creates a new layer in the Docker image with just the changes from the previous



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.

## ANGuments AND ENVIRONMENT VARIABLES IN DOCKER



### **ENVironment VARIABLES – RUNTIME**

* **Available in Dockerfile and in the application code**
* The env variables are set during the “runtime”. We can set the environment variable via **docker run command** or in the Dockerfile

*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**

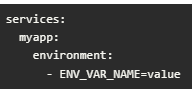
*Replace ENV\_VAR\_NAME with the name of the environment variable we want to set, and value with the desired value.*

1. **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 -1

SETTING THE NODE PORT USING ENV VARIABLE

|  |  |
| --- | --- |
| **SOURCE CODE** | **DOCKER FILE** |
| app.listen(process.env.PORT); | FROM docker.repo1.uhc.com/gov-prog-digital/gpd-node19-alpine:latest  RUN npm config set registry http://repo1.uhc.com/artifactory/api/npm/npm-virtual  WORKDIR /app  COPY package.json .  RUN npm install  COPY . .  **ENV PORT 80**  **EXPOSE $PORT**  CMD [ "node","server.js"] |

SETTING ENV VARIABLE in docker run using “-e”

* In the above docker file -the PORT env variable has been hardcoded – which can be overwritten by “-e” during the “docker run” command

**docker run --rm -d -p 3000:8000 --name feedback-data-volume-app -e PORT=8000 asinha37/feedback-data-volumes**

SETTING ENV VARIABLE USING .env file

|  |  |
| --- | --- |
| **.env FILE** | **COMMAND** |
|  | * In the docker run command we need to specify the path of .env file * Note – the will overwrite the value of env variable in Dockerfile   docker run --rm -d -p 3000:8000 --name feedback-data-volume-app **--env-file ./.env** asinha37/feedback-data-volumes |

#### EXAMPLE -2

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**

### ARGUMENTS – BUILD TIME ARGUMENTS

* The Arguments are specified in the Docker during the build time.
* They are not accessible via command line command not in the application code.

**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 .**

#### EXAMPLE

* *In Dockerfile, we can use the ARG instruction to define the build-time argument and then use the ENV instruction to set it as an environment variable.*
* *Note : We can make use of “ARG” in dockerfile only (not in the application code)*

|  |
| --- |
| **DockerFile** |
| FROM docker.repo1.uhc.com/gov-prog-digital/gpd-node19-alpine:latest  RUN npm config set registry http://repo1.uhc.com/artifactory/api/npm/npm-virtual  WORKDIR /app  COPY package.json .  RUN npm install  COPY . .  **ARG DEFAULT\_PORT=80**  **ENV PORT $DEFAULT\_PORT**  EXPOSE $PORT  CMD [ "node","server.js"] |

* **BUILD COMMAND(during build time ):** This will build the image

**docker build --build-arg DEFAULT\_PORT=8000 -t asinha37/feedback-data-volumes .**

* **RUNNING THE CONTAINER**

**docker run --rm -d -p 3000:8000 --name feedback-data-volume-app asinha37/feedback-data-volumes**

## 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

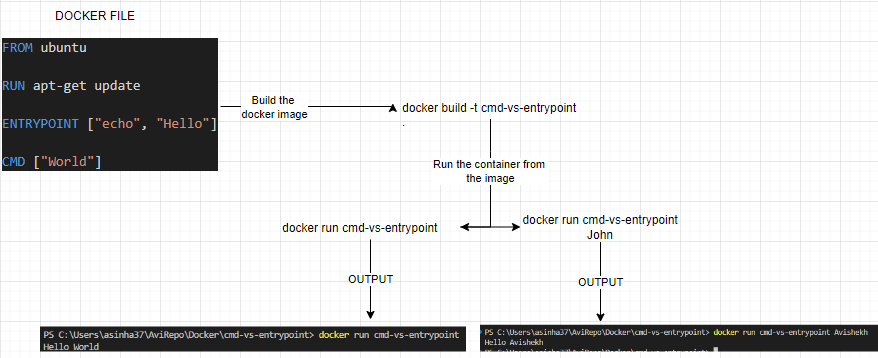
* **The ENTRYPOINT instruction specifies the command that will be executed when the container is started**.
* **ENTRYPOINT is the process that’s executed inside the container.**
* The ENTRYPOINT Dockerfile instruction sets the process that’s executed when the container starts.

**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.**



## DOCKER STORAGE

HOW DOCKER STORES DATA AND HOW IT MANAGES FILE SYSTEMS OF CONTAINERS?

|  |  |
| --- | --- |
|  | **HOW A DOCKER STORES DATA ON THE LOCAL FILE SYSTEM?**  When we install docker - it creates folder structure at **var/lib/docker**. It has multiple subfolders   * **/var/lib/docker/containers**:   + **Contains directories for each running or stopped container on the Docker host**.   + Each container has its own unique directory, which holds metadata and filesystem layers specific to that container. * **/var/lib/docker/image**:   + Stores Docker images and their associated layers.   + Each image has its own directory, and within each image directory, there are subdirectories for each layer. * **/var/lib/docker/volumes**:   + Stores data volumes that are created and managed by Docker.   + Each volume has its own directory, and data within these directories can be mounted into containers. * **/var/lib/docker/network**:   + Contains files and directories related to Docker networking.   + It stores network configuration details, such as bridge networks and network drivers. * **/var/lib/docker/tmp**:   + A temporary directory used by Docker for various operations, such as building images or extracting archives. * **/var/lib/docker/swarm**:   + If you are using Docker Swarm for container orchestration, this directory stores files related to Swarm management. |

### DOCKERS LAYERED ARCHITECTURE

Docker utilizes a layered architecture known as the Docker layered filesystem or Docker image layers. This architecture plays a crucial role in Docker's efficiency and flexibility.

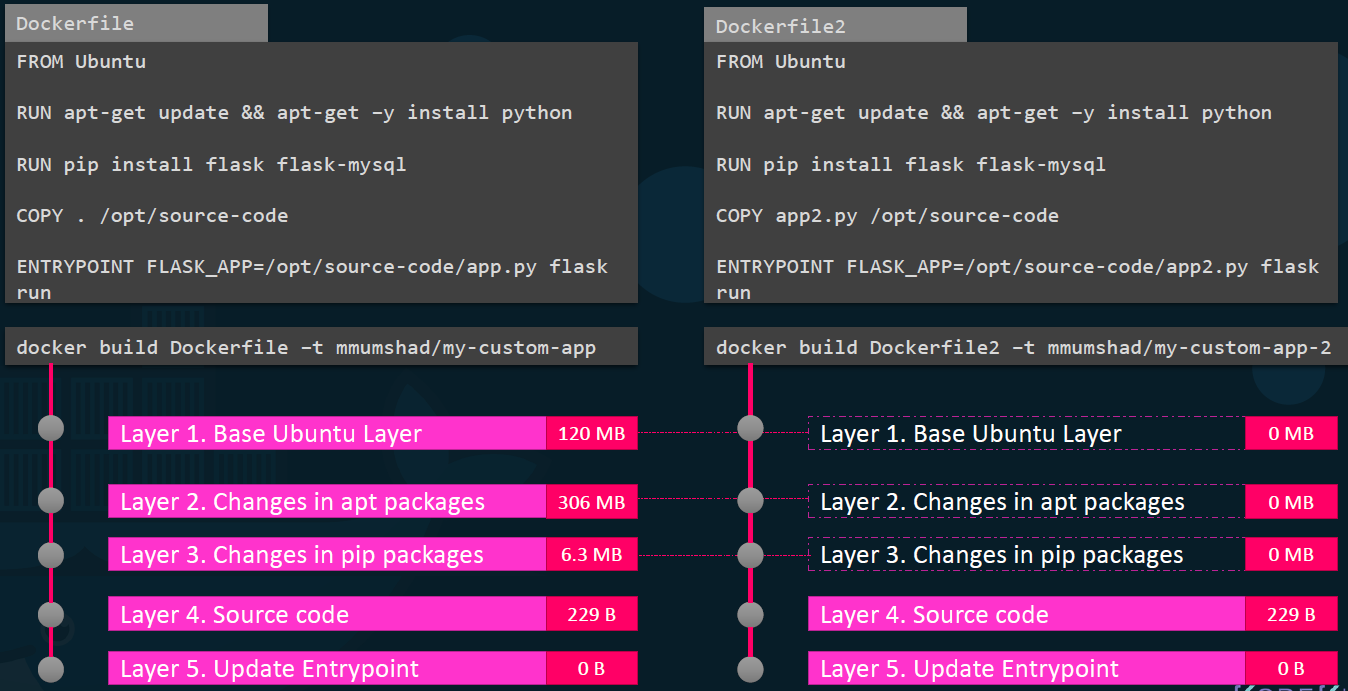
Here's an overview of Docker's layered architecture:

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

* BASE IMAGE
  + At the bottom of the architecture is the base image layer.
  + The base image provides the foundation for the Docker container. It contains the operating system and essential software packages needed to run applications.
* IMAGE LAYERS
* On top of the base image, Docker creates multiple read-only image layers. Each layer represents a specific modification or addition to the base image.
* Layers are created incrementally as changes are made to the container, such as installing packages or modifying files. These layers are stacked on top of each other, forming a chain-like structure.
* COPY-ON-WRITE
  + Docker uses a copy-on-write mechanism to optimize the use of storage space.
  + When a container is started, a read-write layer, known as the container layer or the "writable layer," is added on top of the image layers.
  + This layer allows the container to make changes without modifying the underlying image layers. Instead, it creates new layers for any modifications made to the container, such as creating or modifying files.
* EFFICIENCY AND REUSABILITY
  + The layered architecture provides several benefits.
    - Firstly, it enables efficient storage usage as multiple containers based on the same image can share the same base and intermediate layers, reducing disk space requirements.
    - Secondly, it promotes reusability since layers are cached and can be reused across multiple containers and deployments. This results in faster container creation times and reduces the need to download and store redundant layers.
* IMAGE DISTRIBUTION
  + Docker images are distributed as a series of layers. When pulling an image from a registry, Docker only needs to download the new or updated layers, rather than the entire image. This minimizes network bandwidth and speeds up the image retrieval process.

In the below example:

* Each line of instruction in the docker file creates a new layer in the Docker image with just the changes from the previous layer.

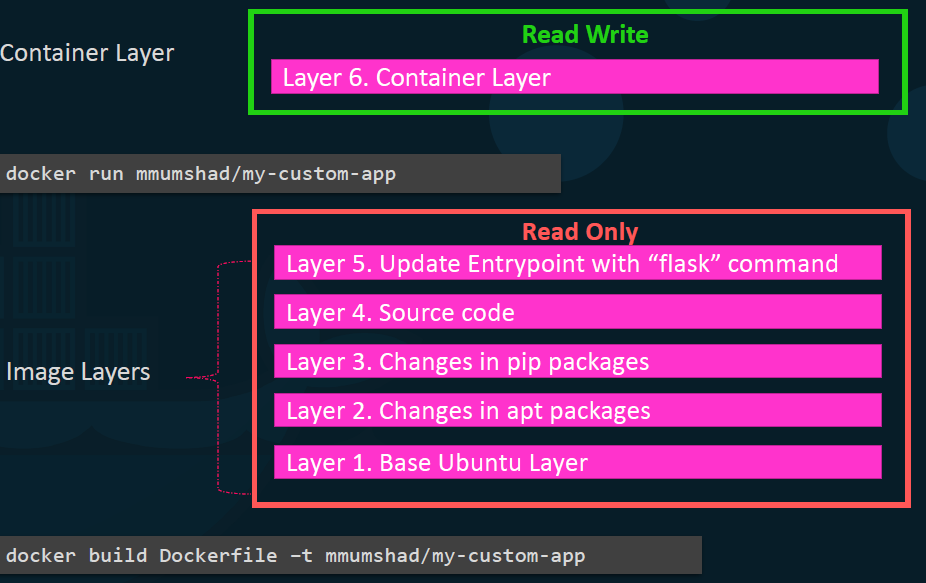


1. The first layer is a base Ubuntu operating system
2. Followed by the second instruction that creates a second layer which installs all the APT packages
3. The third instruction creates a third layer which with the python packages
4. Followed by the fourth layer that copies the source code over.
5. And then finally the fifth layer that updates the entry point of the image

**NOTE - EACH LAYER ONLY STORES THE CHANGES FROM THE PREVIOUS LAYER.**

Let's consider a second application this application has a different docker file but is very similar to our first application as in it uses the same base image as a ubuntu uses as the same python and flask dependencies but uses a different source code to create a different application hence it will have a different entry point.

* When we run the docker build command to build a new image for this application since the first three layers of both the applications are the same Docker is not going to build the first three layers, instead it reuses the same three layers it built for the first application from the cache and only creates the last two layers with the new sources and the new entry point this way Docker builds images faster and efficiently saves disk space.
* This is also applicable if we update our application code whenever we update our application code such as the **app2.py** in this case Docker simply reuses all the previous layers from cache and quickly rebuilds the application image by updating the latest source code thus saving us a lot of time during rebuilds and updates.



Let's rearrange the layers bottom up

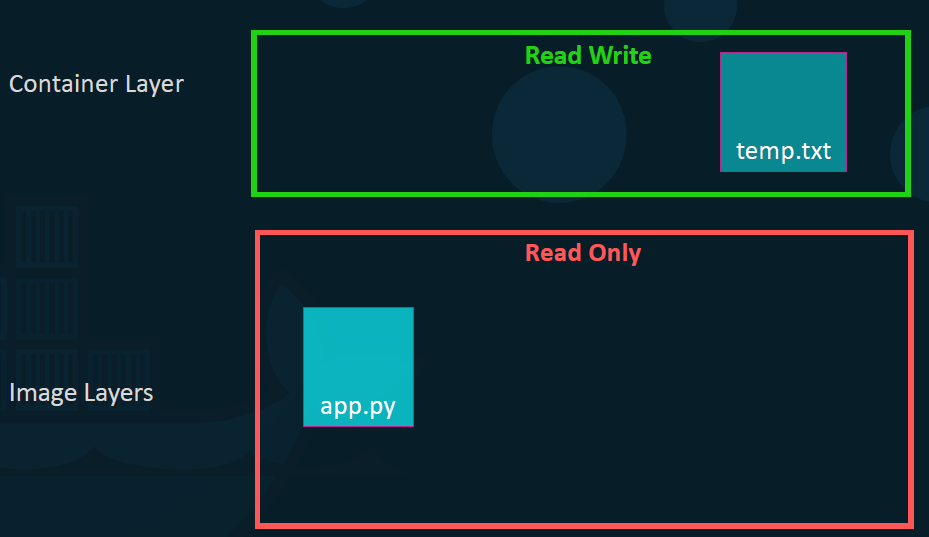
* Bottom we have the base ubuntu layer then the packages then the dependencies and then the source code.

of the application and then the entry point all these layers are created when we run the **docker build**

**command all the image layers put together** to form the final Docker image.

* **Once the build is complete, we cannot modify the contents of these layers and so they are read only and we can only modify them by initiating a new build when we run a container based on this image.**

### COPY ON WRITE

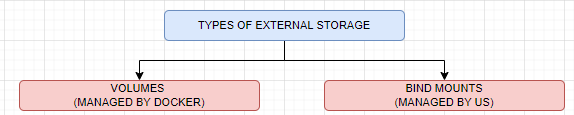


Using the **docker run command** Docker creates a container based off these layers and creates a new writable layer on top of the image layer.

* The writable layer is used to store data created by the container such as log files by the applications, any temporary files generated by the container or just any file modified by the user on that container.
* The life of this layer though is only as long as the container is alive - When the container is destroyed this layer and all the changes stored in it are also destroyed.
* **Since - the same image layer is shared by all containers created using this image if we log into the newly created container and create a new file called temp.txt. Then it would create that file in the container layer which is read and write.**
* In the above example - As the application code is baked is into the image. If we want to modify the this file but before I save the modified file Docker automatically creates a copy of the file in the read-write layer so we end-up modifying a different version of the file in the read write layer.
* All future modifications will be done on this copy of the file in the read write layer. This is called copy on write mechanism
* The image layer being read only just means that the files in these layers will not be modified in the image itself so the image will remain the same all the time until we rebuild the image using the docker build command.
* When the is removed - all the data that was stored in the container layer also gets deleted. i.e.- The change we made to the app.py and the new temp file we created will also get removed.

## DOCKER VOLUME AND BIND MOUNTS

* Volume and bind mounts are both mechanisms used in containerization technologies, such as Docker, to provide access to file systems within containers. However, they differ in how they work and the level of flexibility they offer.



|  |  |
| --- | --- |
| **VOLUME** | **BIND MOUNTS** |
| Image Source | Image Source |

**VOLUMES**

* + **Volumes are managed by the container runtime** and provide a way to persist and share data between containers or between the host machine and containers.
  + **The volumes are completed isolated from the host file system.**

**VOLUMES ARE TYPICALLY MANAGED OUTSIDE THE CONTAINER'S FILE SYSTEM AND OFFER SEVERAL ADVANTAGES:**

* + Data is preserved even if the container is stopped or removed.
  + Multiple containers can share the same volume, allowing for data sharing and collaboration.
  + Volumes can be backed up, restored, and managed independently of containers.
  + Volumes can be easily moved between different hosts or environments.

BIND MOUNTS

* Bind mounts, on the other hand, are a direct mapping of a host file or directory into a container. With bind mounts, a specified file or directory on the host machine is mounted into the container's file system.

**Bind mounts have the following characteristics:**

* + Changes made to files or directories within the bind mount are immediately visible on both the host and the container.
  + Bind mounts can provide more flexibility and control over data access and sharing.
  + Bind mounts can be used to access sensitive host system files and directories.
  + Bind mounts do not have the same level of isolation and portability as volumes.
* Note: In summary, volumes are typically recommended for managing data that needs to be shared or persisted between containers, while bind mounts are useful for accessing specific files or directories on the host machine within a container.

### VOLUME

* Although the volumes are created in the host machine but it cannot be accessed directly in the host machine- as the volumes are managed by docker itself.

|  |  |
| --- | --- |
| Step 1: CREATE A VOLUME | docker volume create <volume\_name>  Example: docker volume create **feedback-volume** |
| STEP 2: ATTACH THE VOLUME TO THE CONTAINER | |
| docker run --rm -p 3000:3000 --name favourite-app -v **<volume\_name>:<path\_in\_the\_container>** asinha37/favourite-node  docker run --rm -p 3000:3000 --name favourite-app -v **feedback-volume:/app/feedback** asinha37/favourite-node | |

|  |  |
| --- | --- |
| **CREATES A NEW VOLUME** | **docker volume create myvolume** |
| **LISTS ALL THE VOLUMES AVAILABLE ON THE DOCKER HOST.**  ***(This will not list the bind mount volumes as they are not managed by docker*)** | **docker volume ls** |
| **PROVIDES DETAILED INFORMATION ABOUT A SPECIFIC VOLUME.**   * The mountpoint refers to the path within a container where a volume is mounted. * When we create or define a volume and mount it to a container, **the mountpoint is the location within the container's file system where the contents of the volume will be accessible.** | **docker volume inspect myvolume** |
| **REMOVES A VOLUME.**  (Before removing the volume – we have first stop and remove the container) | **docker volume rm myvolume** |
| **REMOVES ALL UNUSED VOLUMES.** | **docker volume prune** |
| **MOUNTS A VOLUME TO A CONTAINER.** | **docker run -d -v myvolume:/path/to/mount myimage**  **OR**  **docker run -d --mount source=myvolume,target=/path/to/mount myimage** |
| **COPIES FILES OR DIRECTORIES BETWEEN A CONTAINER AND A VOLUME OR BETWEEN TWO VOLUMES.** | **docker volume cp /path/to/local/file myvolume:/path/to/mount** |

### BIND MOUNTS

* Bind mounts allow us to mount a file or directory from the host machine into a container.
* With bind mounts, we can specify a specific file or directory on the host and mount it at a specific location within the container's file system. This enables sharing and accessing files or directories between the host and the container.
* The data in the bind mounts are editable – *this is the key difference the bind mounts and volumes*

KEY POINTS ABOUT BIND MOUNTS

1. FLEXIBILITY
   1. Bind mounts provide flexibility in choosing the source location on the host machine.
   2. We can mount files or directories from any location on the host's file system into the container, regardless of where the Docker project is located.
2. TWO-WAY SYNC
   1. Bind mounts facilitate a two-way sync between the host and the container.
   2. Any changes made to the files or directories on the host are immediately reflected within the container, and vice versa. This allows for easy data sharing and collaboration between the host and the container.
3. NO COPYING
   1. Unlike volumes, which create a new directory within the Docker environment, bind mounts directly link the container to the host's file system.
   2. This means that the data is not copied into the container. Instead, the container reads and writes directly to the host's file system.
4. PERMISSIONS
   1. When using bind mounts, permissions from the host are applied to the mounted files or directories within the container. This can be advantageous for cases where specific file permissions or ownership need to be maintained.

#### SETTING UP BIND MOUNTS

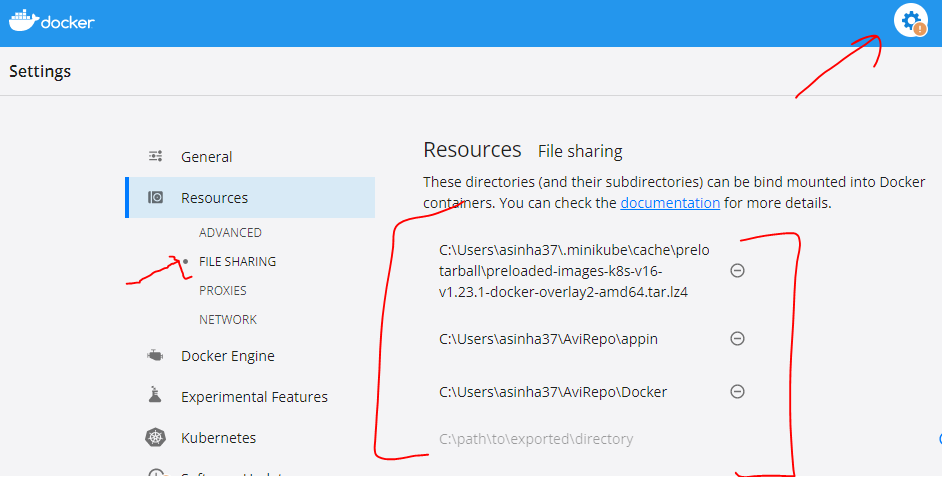
|  |  |
| --- | --- |
|  | Step 1: Create a folder in the host machine(e.g. “data” folder) to which container path has to be mapped  Step 2: Map the path using “-v” option  SYNTAX  docker run --rm -d -p 3002:80 --name feedback-data-volume-app **-v** **<absolute\_path\_in\_host\_machine>:<mapped\_path\_in the container>** asinha37/feedback-data-volumes  EXAMPLE  docker run --rm -d -p 3002:80 --name feedback-data-volume-app **-v C:\Users\asinha37\AviRepo\Docker\data-volumes\data:/app/feedback** asinha37/feedback-data-volumes   * Keep the path in “quotes” – if the path has some special character or space. |

**NOTE**

* If we don't always want to copy and use the full path, we can use these **shortcuts**:

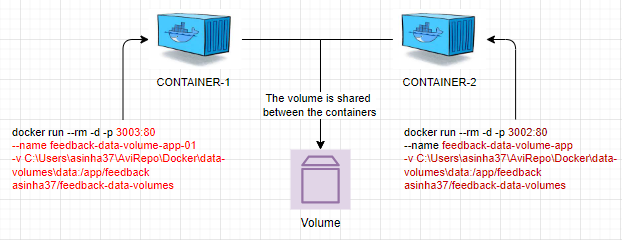
|  |  |
| --- | --- |
| **macOS / Linux** | **-v $(pwd):/app** |
| **Windows** | **-v "%cd%":/app** |

* Docker should have access to folder which we are mapping in the host machine.
* To give permission to the mapped folder
  + Open Docker desktop 🡪 Setting 🡪- Resources 🡪 File Sharing
  + The path should be subfolder of the paths listed in Resources section
  + If the path is not listed in the add it in the existing list



#### SHARING VOLUME BY THE CONTAINER

* In the below example – we started 2 container(with different names) with a same image – using the same image
* The Node app(in containers) is running on different ports(using port mapping)



#### READ ONLY VOLUMES

* We can create read-only volumes to restrict write access to the mounted data within a container i.e. the container will not be able to edit the data in the volume.This can be useful when we want to ensure the integrity and immutability of the data.
* By using read-only volumes, we can prevent accidental modifications to the data within the container. It is important to note that read-only volumes do not prevent modifications from within the host machine. They only restrict write access from within the container.

|  |  |
| --- | --- |
| CREATING A READONLY VOLUME  To create a read-only volume, we can add the **:ro option** when defining the volume in the docker run command. | docker run -d -v myvolume:/container/path:**ro** myimage |
| BIND MOUNT AS READ-ONLY:  If we want to use a bind mount as a read-only volume, we can add the :ro option when defining the bind mount. | docker run -d -v /host/path:/container/path**:ro** myimage |

## DOCKER IGNORE FILE

* A .dockerignore file is a text file that we can create in our project directory to specify which files and directories should be excluded when building a Docker image.
* It works similarly to the .gitignore file used in Git to exclude certain files and directories from version control.

**KEY POINTS ABOUT USING A .DOCKERIGNORE FILE:**

* Creating a .dockerignore File:
  + To create a .dockerignore file, simply create a new text file named .dockerignore in the project directory.
* SYNTAX AND PATTERNS:
  + The syntax and patterns used in a .dockerignore file are similar to those used in a .gitignore file.
  + Each line in the file represents a pattern that specifies which files or directories should be ignored.

Common patterns used in a .dockerignore file include:

1. File or directory names: file.txt, directory/
2. Wildcards: \*.log, \*.tmp
3. Recursive patterns: \*\*/temp/

USAGE:

1. When we build a Docker image using the docker build command, Docker automatically looks for a .dockerignore file in the build context directory and excludes the specified files and directories from the image. This can help reduce the size of the final image and improve build performance.

**SAMPLE DOCKER IGNORE FILE**

# Ignore logs

\*.log

# Ignore temporary files

temp/

# Ignore Node.js modules

node\_modules/

**In this example, all log files, the temp directory, and the node\_modules directory will be excluded from the Docker image during the build process.**

### EXAMPLE -1

|  |  |  |
| --- | --- | --- |
| Dockerfile | .dockerignorefile | FILE STRUCTURE |
| **FROM node**  **WORKDIR /app**  **COPY package.json .**  **RUN npm install**  **COPY . .**  **EXPOSE 80**  **CMD [ "node","server.js"]** | **node\_modules/** |  |

1. In the above example – when we run the node project in local machine- we need to perform npm install 🡪 which in turn creates the node\_module folder
2. Now – in the Dockerfile – we are copying all the files from the source to “image”- which includes node\_module too
3. As docker is anyway executing the “npm install” (which will re-create the node\_modules folder OR overwritten by local node\_module folder). Hence copying the node module folder in redundant.
4. Hence – “node\_module” folder is a candidate – to be ignored, while building the image

### EXAMPLE -2

|  |  |
| --- | --- |
| We can add more "to-be-ignored" files and folders to the .dockerignore file. | For example, consider adding the following to entries:  **Dockerfile**  **.git**  This would ignore the Dockerfile itself as well as a potentially existing .git folder |

## DOCKER NETWORKING

* The request/communication from the container too WWW(internet)

### COMMUNICATION BETWEEN CONTAINER AND HOST

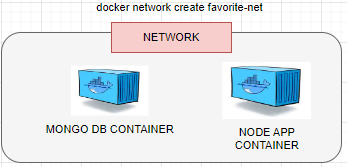
|  |  |
| --- | --- |
|  | * Let consider a case when the container(node app) wanted to access the MongoDB on the host machine. * The container can the address of the of the host machine using “**host.docker.internal**” * Sample Code   mongoose.connect(  'mongodb://**host.docker.internal**:27017/swfavorites',  ....  ); |

### COMMUNICATION BETWEEN CONTAINERS

* Container to container communication can happen by using the IP addesss.

|  |  |
| --- | --- |
| mongoose.connect(  'mongodb://**172.17.0.2:27017**/swfavorites',  { useNewUrlParser: true ,  useUnifiedTopology: true },  (err) => {  if (err) {  console.log(err);  } else {  app.listen(3000);  }  }  ); | * Lets consider a case – When we have to establish a communication between two running container. For example – a Node App and Mongo Db container.  1. **Step 1**: Run the mongo Db container : docker run -d –name mongodb mongo 2. **Step 2**: Inspect the mongdb container to get the IP address. |

* This way of setting up connection is cumbersome as we have to update the IP addess to the connecting application , whenever the mongodb IP address changes



* The communication between the container can happen if they are placed in the same network.

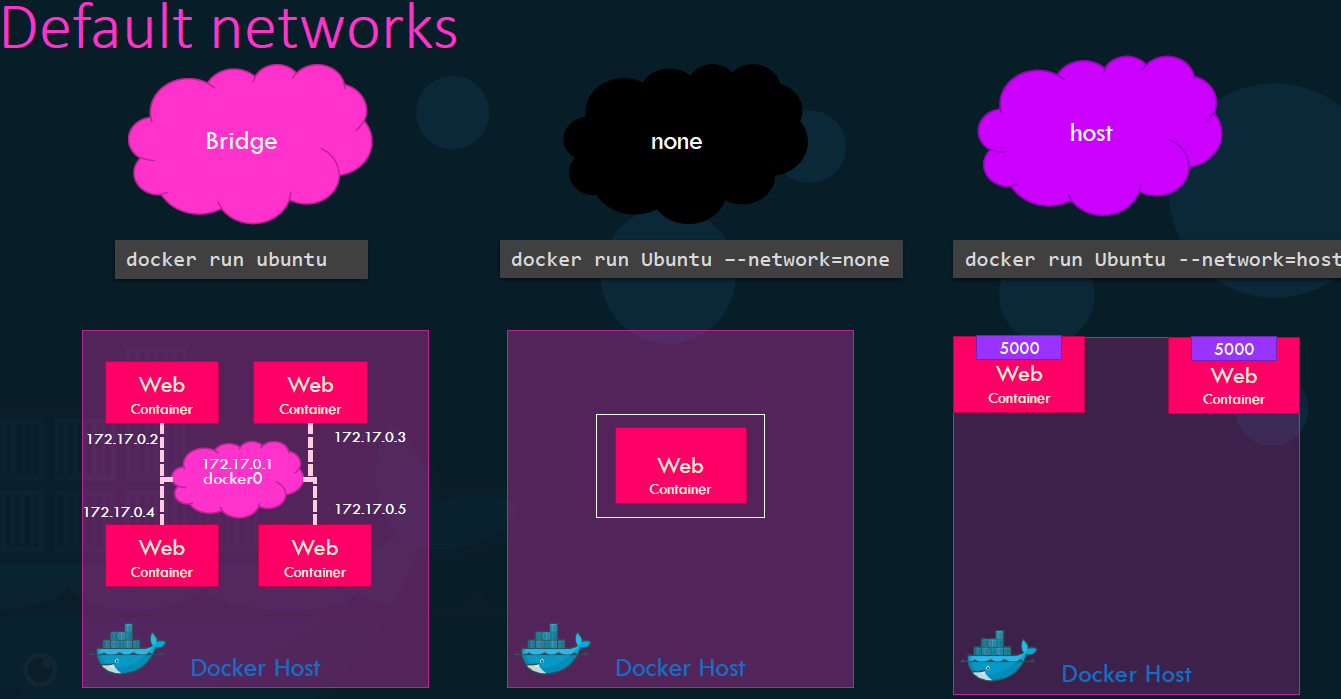
|  |  |
| --- | --- |
| Step 1: CREATE A NETWORK  (These networks are called bridge network) | **docker network create <network\_name>**  **docker network create favorite-net** |
| Step 2: VIEW THE CREATED NETWORK | **docker network ls** |
| **Step 2: ASSOCIATE THE CONTAINER WITH THE NETWORK** | |
| **MONGODB CONTAINER: docker run -d --name mongodb --network favourite-net mongo**  **NODE APP CONTAINER: docker run --rm -p 3000:3000 --name favourite-app --network favourite-net asinha37/favourite-node** | |
| When the container belong to the same network – it can communicate via the container name  **mongoose.connect(**  **'mongodb://mongodb:27017/swfavorites',**  **….);** | |

:

### DOCKER NETWORK DRIVERS

* Docker Networks support different kinds of "**Drivers**" which influence the behavior of the Network.
  + The default driver is the "**bridge**" driver - it provides the behavior shown in this module (i.e. Containers can find each other by name if they are in the same Network).

|  |  |
| --- | --- |
| * The driver can be set when a Network is created, simply by adding the --driver option. | **docker network create --driver bridge my-net**   * **Note:** ***if we want to use the "bridge" driver, we can simply omit the entire option since "bridge" is the default anyways.*** |



When we install Docker, it creates three networks automatically. These are the default network a container gets attached to

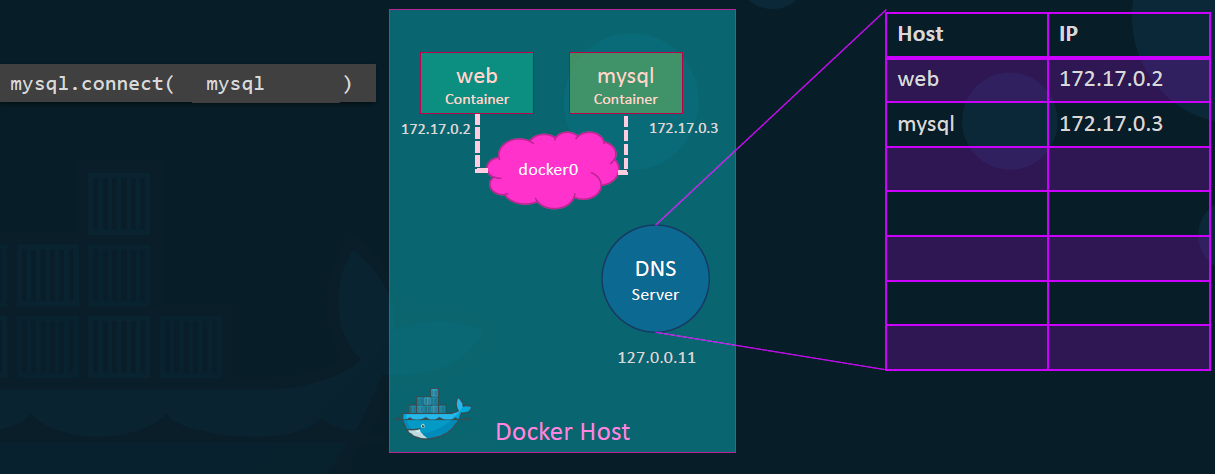
1. **BRIDGE** 
   1. A bridge network is a default network that allows communication between containers on the same Docker host.
   2. When we create a new container without specifying a network, it is automatically connected to the bridge network.
2. **NONE**
3. **HOST BRIDGE**

#### BRIDGE NETWORK

* A bridge network is a default network.
* When we create a new container without specifying a network, it is automatically connected to the bridge network.
* The bridge network allows communication between containers on the same Docker host.

KEY FETURES

* DEFAULT BRIDGE NETWORK
  + When Docker is installed, it creates a default bridge network named "bridge" on the host.
  + This network is used for communication between containers on the same Docker host.
* IP ADDRESS ASSIGNMENT
  + Each container connected to the bridge network gets an IP address assigned from the bridge network's IP address range.
  + By default, Docker uses the CIDR notation "**172.17.0.0/16**" for the bridge network, and each container receives an IP address within this range.
* CONTAINER DISCOVERY
  + **Containers connected to the bridge network can communicate with each other using their container names as hostnames**.
  + Docker provides an embedded DNS server that allows containers to resolve other container names to their respective IP addresses within the bridge network.



* PORT MAPPING
  + The bridge network enables port mapping between the container and the host machine. Using the "-p" or "--publish" flag when running a container, we can map container ports to specific ports on the host machine.
  + This allows external services to access the container's exposed ports through the host machine's IP address and mapped ports.
* CONNECTIVITY TO EXTERNAL NETWORKS
  + Containers connected to the bridge network can access external networks, such as the internet, through Network Address Translation (NAT) performed by the Docker daemon.
  + The bridge network acts as an intermediary between the container and the external network, allowing outbound traffic from the container.
* CUSTOM BRIDGE NETWORKS
  + Besides the default bridge network, we can create custom bridge networks using the "**docker network create**" command.
  + Custom bridge networks provide isolation and segmentation between containers and enable us to define specific network settings. Containers can be connected to multiple bridge networks simultaneously.

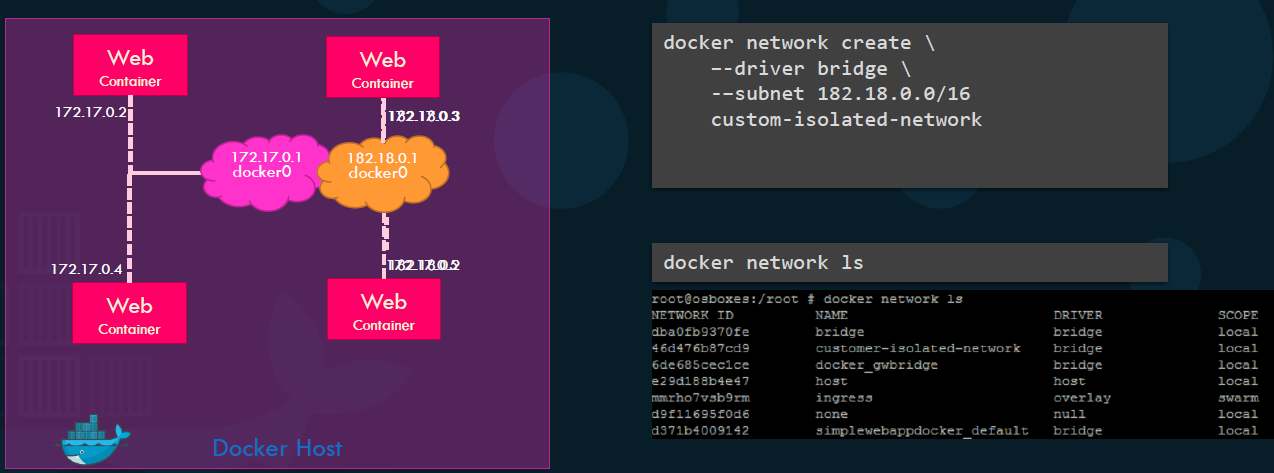
#### HOST NETWORK

* **The host network mode allows a container to use the host machine's networking stack directly.**
* When a container runs in host network mode, it shares the network namespace with the host machine, giving it full access to the host's network interfaces and exposing the container directly to the host's network.

KEY POINTS TO UNDERSTAND ABOUT THE HOST NETWORK MODE IN DOCKER

* NETWORK NAMESPACE
  + By default, Docker containers have their own isolated network namespace, which provides network isolation and allows containers to have their own IP addresses and network configurations. However, when a container is run in host network mode, it uses the same network namespace as the host machine, meaning it shares the host's networking stack.
* PORT BINDING
  + When a container is running in host network mode, it can bind to host ports directly, without any port mapping.
  + This means that any services or applications running in the container can be accessed using the host machine's IP address and the container's port.
* NETWORK ISOLATION
  + Running a container in host network mode removes the network isolation provided by Docker's default networking. It means that the container by-passes Docker's network stack and operates as if it were running directly on the host machine.
  + This can potentially compromise the isolation and security of the container, as it has direct access to the host's network interfaces.
* CONTAINER NAMING:
  + When running a container in host network mode, the container still has its own hostname, which can be used for inter-container communication within the same Docker host. However, portability may be limited if the container relies on specific IP addresses or network configurations of the host machine.
* COMPATIBILITY AND PERFORMANCE
  + Running a container in host network mode can be useful in certain scenarios where direct access to the host's network stack is required, such as when the container needs to bind to privileged ports (ports below 1024).
  + It can also provide better performance for certain network-intensive applications that require maximum network throughput or low latency.

#### USER DEFINED NETWORK



* Note – By default Docker creates just one default bridge network with the network id 172.17.0.1. All containers associated to this default network will be able to communicate to each other.
* To create our own internal network (to isolate the containers within the docker host) .*For example the first two web containers on internal network 172 and the second two containers on a different internal network like 182*

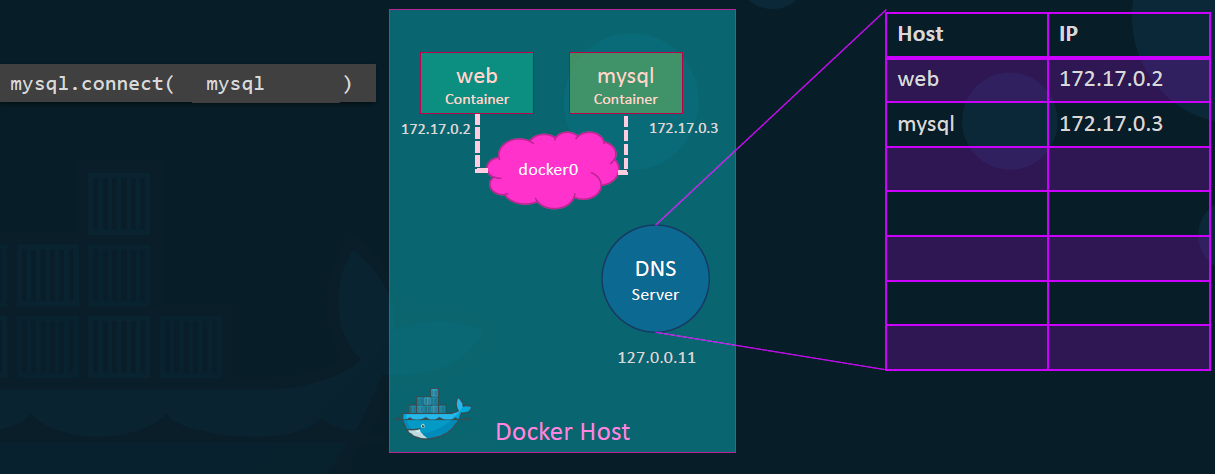
**docker network create [OPTIONS] NETWORK\_NAME**

**Example:**

**docker network create --driver=bridge --subnet=172.18.0.0/16 --gateway=172.18.0.1 --ip-range=172.18.0.0/24 my-network**

|  |  |
| --- | --- |
| --driver | * Specifies the driver to be used for the network. * Docker provides various drivers such as bridge (default), overlay, macvlan, host, etc. The bridge driver is typically used for most scenarios |
| --subnet | * Specifies the subnet in CIDR notation for the network. For example, --subnet=172.18.0.0/16. |
| --gateway | * Specifies the gateway IP address for the network. For example, --gateway=172.18.0.1. |
| --ip-range | * Specifies the IP address range within the subnet to be used for dynamic IP assignment to containers. For example, --ip-range=172.18.0.0/24 |
| --attachable | * Allows other containers to attach to the network after it has been created. * By default, only newly created containers can join the network |
| --label | * Adds metadata labels to the network. Labels can be used for filtering and organizing networks. |

### EMBEDDED DNS



* In Docker, embedded DNS (Domain Name System) is a built-in feature that enables name resolution between containers within the same Docker network.
* It allows containers to communicate with each other using their container names as hostnames.

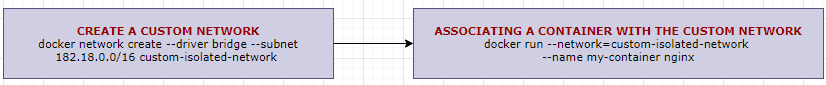
#### HOW EMBEDDED DNS WORKS IN DOCKER?

* **When we create a custom network using the docker network create command, Docker automatically assigns an embedded DNS server to that network. This DNS server is responsible for resolving container names to their respective IP addresses within the network.**
* Each container connected to the custom network is assigned a unique hostname based on its container name. The embedded DNS server maintains a mapping of container names to IP addresses.
* Containers within the same network can communicate with each other using their container names as hostnames. For example, if we have a container named "**container1**" and another container named "**container2**" on the same network, "container1" can send requests to "container2" using "container2" as the hostname.
* Docker automatically updates the embedded DNS server's records whenever containers are added or removed from the network. This ensures that the DNS server always has an up-to-date mapping of container names to IP addresses.
* Hence by leveraging embedded DNS in Docker, we can simplify container communication within a network, as containers can refer to each other using their container names instead of hardcoding IP addresses.

### ASSOCIATING A CONTAINER TO A NETWORK

To associate the container with any other network we specify the network information using the network command line parameter.

|  |  |
| --- | --- |
| **docker run ubuntu** | * If no network parameter is specified, the containers are attached to “bridge” network by default |
| **docker run ubuntu –network=none** | Attached to “none” network |
| **docker run ubuntu –network=host** | Attached to “host” network |



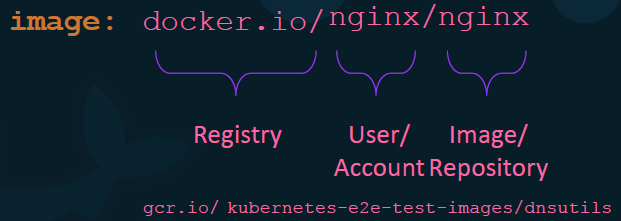
## DOCKER REGISTRY

* A Docker registry is a central repository for storing and distributing Docker images.
* It is a service that allows users to upload, download, and manage Docker images, which are the building blocks for creating Docker containers.
* The default public Docker registry is Docker Hub (hub.docker.com). Docker Hub also serves as the default registry for Docker Engine unless otherwise specified.

Some popular alternatives to Docker Hub for private registries include:

1. AZURE CONTAINER REGISTRY
2. AMAZON ELASTIC CONTAINER REGISTRY (ECR)
3. GOOGLE CONTAINER REGISTRY

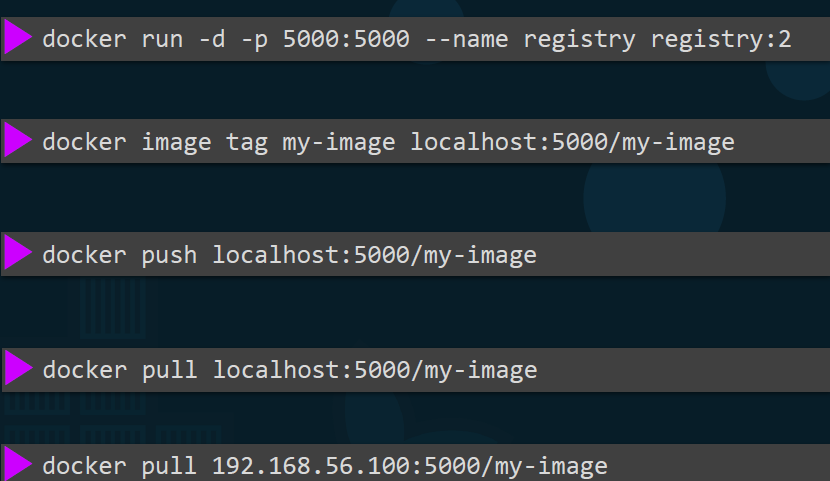
### IMAGE NAMES IN REGISTRY

****

### RUNNING A PRIVATE DOCKER REPOSITORY

|  |  |
| --- | --- |
| To login to private registry | **docker login private-registry.io** |
| To run the image in private registry | docker run **private-registry.io/internal-app** |

### DEPLOYING PRIVATE REGISTRY IN ORGANIZATION



1. STEP 1: PULL THE DOCKER REGISTRY IMAGE :
   1. To create private repository. Since the docker registry itself is available as an image: **docker pull registry**
2. STEP 2: RUN A REGISTRY CONTAINER.
   1. For example, we’ll run it with the name registrydev

**docker run -d -p 5000:5000 --restart=always –name registrydev registry:2**

1. STEP 3: TAG THE IMAGE

For example, to tag the image (say ubuntu image)

1. **docker pull ubuntu:16.04**
2. **docker tag ubuntu:16.04 localhost:5000/myubuntu:16.04**
3. STEP 4: PUSH THE IMAGE

**docker push localhost:5000/myubuntu:16.04**

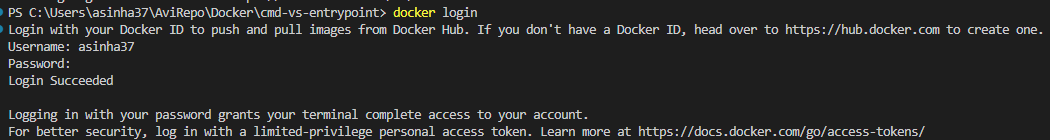
Now its time to push some images to our registry server. Let's push two images for now .i.e. nginx:latest and httpd:latest.

Note: Don't forget to pull them first.  
To check the list of images pushed , use curl -X GET localhost:5000/v2/\_catalog

### 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.



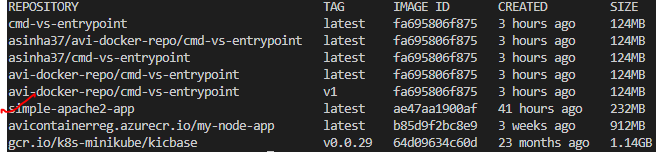
1. **TAG IMAGE**:
   1. Before pushing the image, ensure that it is properly tagged with your Docker Hub username and the repository name.
   2. Use the docker tag command to tag your image.

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

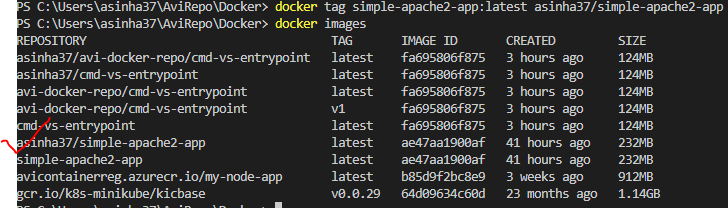
*Replace your-docker-username with your Docker Hub username and your-repository-name with the desired repository name*

**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**

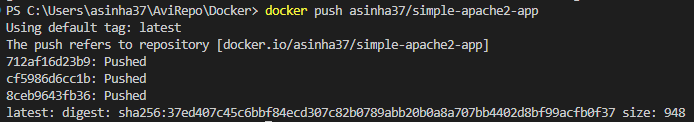


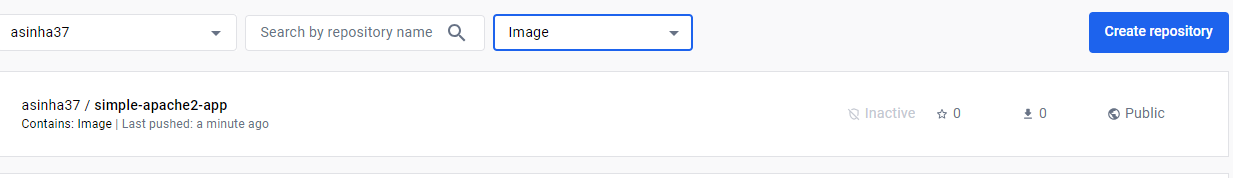
1. PUSH THE IMAGE: Use the docker push command to push the tagged image to Docker Hub.

**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





## LINKING THE CONTAINER – LEGACY METHOD

* The --link flag in Docker is used to create a network link between containers.
* It allows one container to access another container's network services using a defined alias.

**The syntax for the --link flag is as follows: docker run --link <container-name-or-id>:<alias> ...**

Here's how it works:

Start a container that you want to link to another container. For example, if you have a container named "db" running a database service, you can link it to another container:



docker run --name app --link db:database my-image

In this example, the app container is being linked to the db container using the alias database.

Inside the linked container (app in this example), environment variables are automatically created based on the link information. These environment variables provide access to the linked container's network services. Specifically, Docker sets the following environment variables:

DATABASE\_NAME: The hostname or IP address of the linked container.

DATABASE\_PORT: The exposed port number(s) of the linked container.

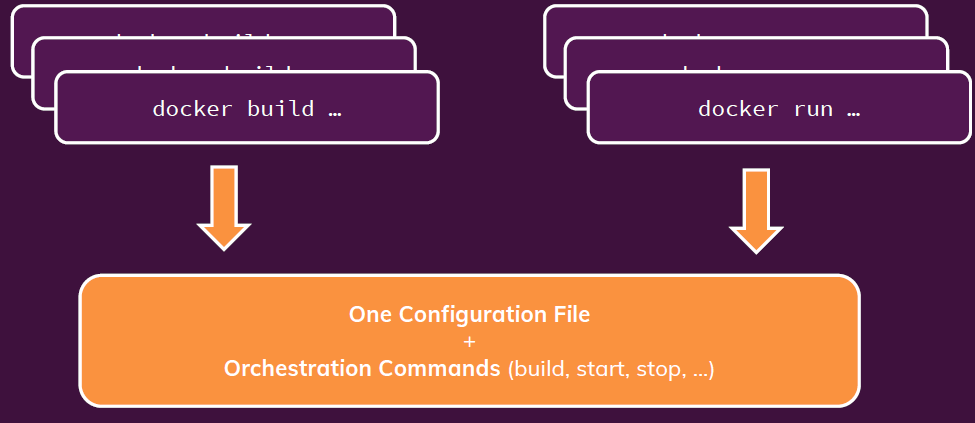
DATABASE\_ENV\_<variable>: Any environment variables prefixed with DATABASE\_ENV\_ from the linked container.

For example, if the linked container (db) has an environment variable named DB\_USER, it will be available in the linking container (app) as DATABASE\_ENV\_DB\_USER.

You can use these environment variables in your application code to connect to the linked container's network services.  
  
Please note that the --link flag is considered a legacy feature in Docker, and it's recommended to use user-defined networks for container communication instead. User-defined networks provide more flexibility and control over container linking.

## DOCKER COMPOSE - MULTI-CONTAINER ORCHESTRATION

* 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.



NOTE

* Docker compose does NOT replace Dockerfiles for custom images.
* Docker compose NOT suited for managing multiple containers of different hosts(machine)

### WRITING THE DOCKER COMPOSE FILE

|  |  |
| --- | --- |
|  | * In docker – we define services (which in the end can be translated with containers) that make up the multi-container application. * **Below every service. We can define - Port should be published, Environment variables the container might need, volumes that should be assigned to this container and assign networks to the container** |

VERSION IN docker-compose.yaml

* In a Docker Compose YAML file, we can specify the version of the Docker Compose file format we are using.
* This version declaration helps ensure compatibility and allows us to use specific features and syntax available in that version.
* STEP 1 : DEFINE YOUR DOCKER COMPOSE YAML FILE:
  + Create a new file named docker-compose.yml in the project directory.
  + This file will define the services, networks, and volumes.
* STEP 2 : DEFINE SERVICES:

|  |  |
| --- | --- |
| * In the docker-compose.yml file, start by defining the services for our application. * **Each service should have a name, an image or build context,** and any additional configuration options we need. * For example, a basic service definition for a web application might look like this: | **version: '3'**  **services:**  **web:**  **image: nginx:latest**  **ports:**  **- 8080:80** |

* STEP 3 : ADD NETWORKS AND VOLUMES (OPTIONAL)

|  |  |
| --- | --- |
| * If the application requires additional networks or volumes, we can define them in the docker-compose.yml file. * For example, to create a custom network and a volume, we can add the following sections: | **version: '3'**  **services:**  **web:**  **image: nginx:latest**  **ports:**  **- 8080:80**  **networks:**  **mynetwork:**  **volumes:**  **myvolume:** |

* STEP 4: CUSTOMIZE CONFIGURATIONS
  + We can customize the configurations for each service by adding environment variables, mounting volumes, setting up dependencies, or defining health checks.
* STEP 5: BUILD AND RUN THE APPLICATION
  + Once we defined services and configurations in the docker-compose.yml file, we can build and run the application using the Docker Compose CLI.
  + Navigate to the directory containing the docker-compose.yml file, and run the following command

|  |  |
| --- | --- |
| **docker-compose up -d** | This command will build the necessary containers and start the application in detached mode. |
| **docker-compose up -d –build** | This option **forces a rebuild of the Docker images before starting the containers**. **It ensures that any changes made to the Dockerfile or build context are applied** |
| **docker-compose down** | It stops and remove the containers and removes network. This does not remove the volume. |
| **docker-compose down -v** | To delete volume as well: |
| **docker-compose logs** | To view the container logs. |

* **We can add the -d flag to run the containers in detached mode (in the background).**

#### SAMPLE DOCKER COMPOSE

|  |  |
| --- | --- |
| version: "3.8"  services:  mongodb:  image: 'mongo'  volumes:  - data:/data/db  #environment:  #- MONGO\_INITDB\_ROOT\_USERNAME: max  #- MONGO\_INITDB\_ROOT\_PASSWORD: secret  #- MONGO\_INITDB\_ROOT\_USERNAME=max  #- MONGO\_INITDB\_ROOT\_PASSWORD=secret  env\_file:  - ./env/mongo.env  # backend:  # frontend:  volumes:  data: | ENVIRONMENT VARIABLES   * Environment variables can be defined either in docker-compose.yaml or in a separate “env” file * The environment variables in docker-compose file can be provided using “:” or “=” – with a key value pair * In case of separate env file – we configure the path of env file for “env\_file” property.      * The docker compose command create a default network(because no explicit network has been configured ) and the volume. |

##### NETWORKING IN DOCKER COMPOSE

* Networking in Docker Compose allows us to define and manage the communication between different containers within the Docker application.
* **It enables containers to discover and communicate with each other using their service names, making it easier to create and manage complex multi-container applications**.

**EXAMPLE**

|  |  |
| --- | --- |
| version: "3.8"  services:  service1:  image: image1  networks:  - network1  - network2  service2:  image: image2  networks:  - network1  networks:  network1:  network2: | * In this example, there are two services (service1 and service2) and two networks (network1 and network2).   + service1 is connected to both network1 and network2.   + service2 is connected only to network1. * By default, Docker Compose creates a default network for the application if we don't specify any networks explicitly. * **When containers are connected to the same network, they can communicate with each other using their service names as hostnames**. For example, service1 can connect to service2 by using the hostname service2 within its code. * Networking in Docker Compose also allows us to control the visibility of services between networks. By default, services within a network can communicate with each other, but you can limit the visibility of services by specifying the networks they are connected to. |

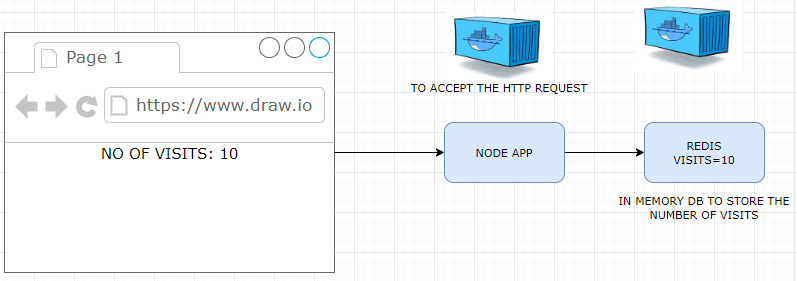
##### VOLUMES IN DOCKER COMPOSE

* Volumes in Docker Compose are used to persist data and share files between containers and the host machine.
* They allow us to store and manage data separately from the container itself, making it easier to manage stateful applications.
* In Docker Compose, we can define volumes using the volumes section within the Docker Compose file. There are different types of volumes we can use:

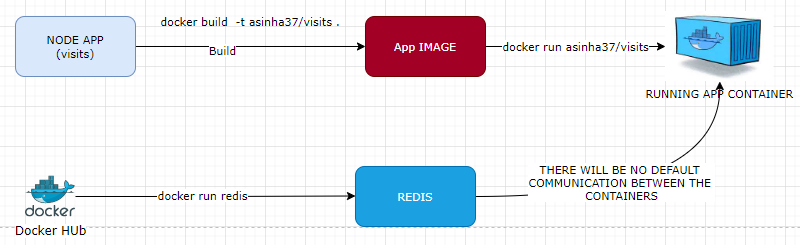
|  |  |
| --- | --- |
| **NAMED VOLUMES**:   * Named volumes are the recommended way to manage persistent data in Docker Compose. * They are created and managed by Docker and can refer to them by name. * *In this example, a named volume called data is created. The service1 container mounts this volume to the /path/to/mount directory inside the container. This allows data to be persisted even if the container is restarted or recreated.* | EXAMPLE  version: "3.8"  services:  service1:  image: image1  volumes:  - data:/path/to/mount  volumes: 🡨 ***on the root level we need to just list the named volumes.***  data: |
| **HOST BIND MOUNTS**:   * Host bind mounts allow us to mount a directory or file from the host machine into the container. * *In this example, the /path/on/host directory on the host machine is mounted to the /path/to/mount directory inside the service1 container.* | version: "3.8"  services:  service1:  image: image1  volumes:  - /path/on/host:/path/to/mount |
| **ANONYMOUS VOLUMES**:   * Anonymous volumes are created and managed by Docker, but they are not given a specific name. They are typically used when we don't need to reference the volume by name * *In this example, an anonymous volume is created and mounted to the /path/to/mount directory inside the service1 container.* | version: "3.8"  services:  service1:  image: image1  volumes:  - /path/to/mount |

### DOCKER COMPOSE PROJECT

* The purpose of the application is to make a Docker container that contains a web application that displays inside the browser the number of times that someone has visited this server.



1. Web server (Node JS App): Which will respond to HTTP requests and generate some HTML to show inside the browser.
2. Redis server (in-memory data store): The purpose of the Redis server is to store the number of times that the page has been visited.



|  |  |  |
| --- | --- | --- |
| NODE APP | | |
| Index.js | DOCKER COMPOSE | package.json |
| **const express = require('express');**  **const redis = require('redis');**  **const app = express();**  **const client = redis.createClient({**  **host: redis-server**  **});**  **client.set('visits', 0);**  **app.get('/', (req, res) => {**  **client.get('visits', (err, visits) => {**  **res.send('Number of visits is ' + visits);**  **client.set('visits', parseInt(visits) + 1);**  **});**  **});**  **app.listen(8081, () => {**  **console.log('Listening on port 8081');**  **});** | **version: '3'**  **services:**  **redis-server:**  **image: 'redis'**  **node-app:**  **build: .**  **ports:**  **- "4001:8081"** | **{**  **"dependencies": {**  **"express": "\*",**  **"redis": "2.8.0"**  **},**  **"scripts": {**  **"start": "node index.js"**  **}**  **}** |
| **DOCKER FILE**  **FROM node:alpine**  **WORKDIR /app**  **ENV npm\_config\_registry=http://repo1.uhc.com/artifactory/api/npm/npm-virtual**  **COPY ./package.json ./**  **RUN npm install**  **COPY . /app**  **CMD ["npm", "start"]** | | |

* As we have a Node application in one container and the Redis application in the separate Docker container.
* Now, these two containers do not have any automatic communication between them (they are two isolated processes)
* To make sure that our Node app can communicate with the Redis server and store information, we need to set up some networking infrastructure between them.
* For such multi container application management like the configuration for the **application's services, networks, and volumes** – We make use of Docker Compose

### STEPS TO SET UP DOCKER COMPOSE

1. Create a docker-compose.yml file in the root path of the project.

SERVICES IN DOCKER COMPOSE

* A service represents a containerized component of the application stack.
* It can be a web server, a database, a caching layer, or any other service that your application requires.

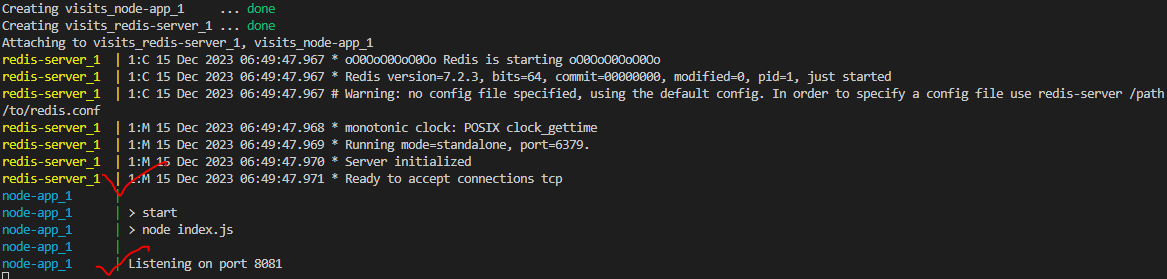
|  |  |
| --- | --- |
| **version: '3'**  **services:**  **redis-server:**  **image: 'redis'**  **node-app:**  **build: .**  **ports:**  **- "4001:8081"** | * We have configured 2 services “redis-server” and “node-app” * node-app is a custom image   + build : Path of docker image   + ports: Port mapping [**host-machine-port:container-port**] * Note – the communication between redis and node-app – we need to define the redis service name in the node app   **const client = redis.createClient({**  **host: redis-server**  **});** |

RUNNING THE APPLICATION USING DOCKER COMPOSE

|  |  |
| --- | --- |
| **docker-compose up** | * This command will start the defined services and create any necessary containers. You will see logs from the containers in the terminal. * Make use of -d option to start the containers in detached mode(run in background) |
| **docker-compose up –build** | **This command will Builds the images🡪Creates the containers🡪Starts the containers🡪Streams the container logs** |
| **docker-compose down** | This will gracefully stop the containers and remove any associated resources. |

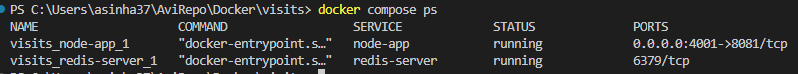
* **The below logs show that the docker compose set up a network to the communication between the containers**





STATUS OF CONTAINERS USING DOCKER COMPOSE

* To view the status of the containers – we make use of : **docker-compose ps**
* Note: This must be executed from the project’s root directory – where we have the docker-compose.yml file



### MAINTANING CONTAINER USING DOCKER COMPOSE

#### CONTAINER RESTART POLICIES

* Docker provides several options for configuring automatic container restart policies.
* These policies determine how Docker handles restarting containers when they exit or encounter errors.

**MOST COMMONLY USED RESTART POLICIES**

|  |  |
| --- | --- |
| **“no”** | * This is the default restart policy. * It means containers will not be automatically restarted, regardless of the exit status or error encountered. * We can manually start the container again. * restart: “no” 🡨 Here “no” has to be in double / single quotes. As “no” is considered as “false” in yaml |
| **always** | * With this policy, Docker will always restart the container, regardless of the exit status. * This is useful for critical services that need to be running continuously. |
| **on-failure** | * This policy specifies that Docker will restart the container only if it exits with a non-zero exit status. * We can also set the maximum number of restart attempts using the **--restart-max-retries** flag. |
| **unless-stopped** | * **When this policy is set, Docker will always restart the container unless we explicitly stop it.** * It ensures that the container starts automatically during system boot or Docker daemon restarts. |

* To specify the restart policy when running a container, you can use the --restart flag with the docker run command:

**docker run --restart=always my-container**

**RESTART POLICY IN DOCKER COMPOSE**

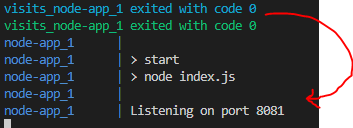
|  |  |
| --- | --- |
| **services:**  **my-service:**  **restart: always** | * We can also set the restart policy in a Docker Compose file using the restart key under the services section. * This sets the restart policy for the my-service container to "always" in the Docker Compose file. |

##### RESTART “ALWAYS” EXAMPLE

* Step 1: To simulate the failure/ crash – change the express code and configure the restart policy in the docker-compose file

|  |  |
| --- | --- |
| Express Code | docker-compose.yml |
| const express = require('express');  const redis = require('redis');  **const process = require('process');**  const app = express();  const client = redis.createClient({  host: 'redis-server'  });  client.set('visits', 0);  app.get('/', (req, res) => {  **process.exit(0);**  client.get('visits', (err, visits) => {  res.send('Number of visits is ' + visits);  client.set('visits', parseInt(visits) + 1);  });  });  app.listen(8081, () => {  console.log('Listening on port 8081');  }); | version: '3'  services:  redis-server:  image: 'redis'  node-app:  build: .  restart: always  ports:  - "4001:8081" |

* Step 2: Start the container using docker compose: **docker-compose up –build**
* **Step 3:** Access the Express Route : <http://localhost:4001/>
* **Step 4:** This will restart the node-app container.



##### RESTART “on-failure” EXAMPLE

Note:

* This policy specifies that Docker will restart the container only if it exits with a non-zero exit status.

|  |  |
| --- | --- |
| Express Code | docker-compose.yml |
| ….  app.get('/', (req, res) => {  **process.exit(1); 🡨 ANY NON ZERO VALUE**  client.get('visits', (err, visits) => {  res.send('Number of visits is ' + visits);  client.set('visits', parseInt(visits) + 1);  });  });  … | version: '3'  services:  redis-server:  image: 'redis'  node-app:  build: .  restart: on-failure  ports:  - "4001:8081" |

* <https://docs.docker.com/compose/>
* <https://docs.docker.com/engine/reference/commandline/compose/>
* <https://github.com/dockersamples/example-voting-app>

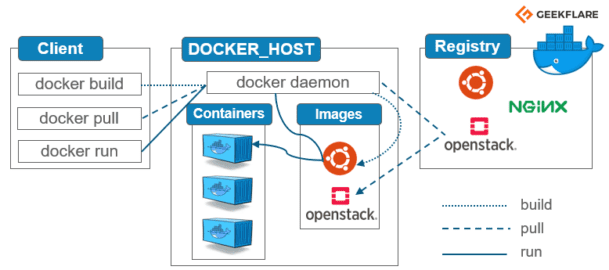
## DOCKER ENGINE

* Docker Engine is the core component of the Docker platform that enables the creation, deployment, and management of containerized applications.
* It is an open-source container runtime that runs on various operating systems, including Linux, Windows, and macOS.

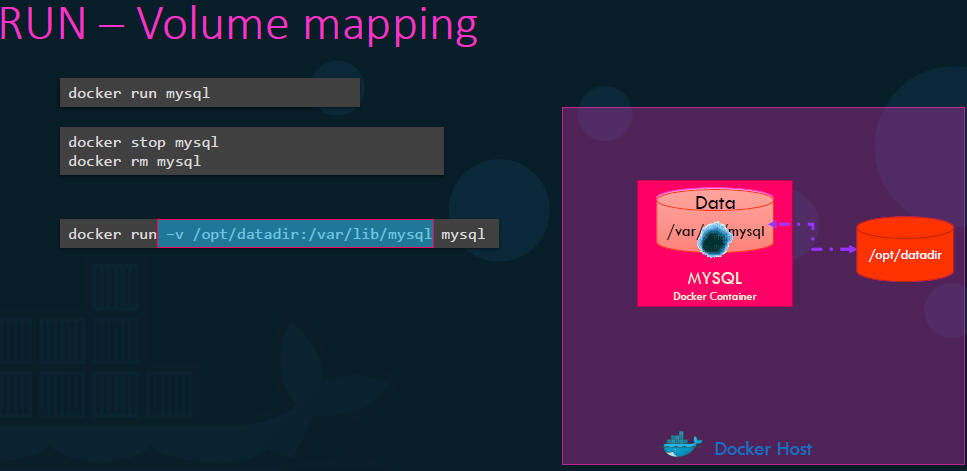
### COMPONENTS OF DOCKER ENGINE

* When we install Docker on a Linux host we are actually installing 3 different components.
  + **THE DOCKER DEMON**
  + **THE REST API SERVER**
  + **DOCKER CLI**
* The DOCKER DEMON is a background process that manages Docker objects such as the images containers volumes and networks.
* The DOCKER REST API SERVER is the API interface that programs can use to talk to the demon and provide instructions.
* DOCKER CLI
  + Is the command line interface to perform actions such as running a container stopping containers destroying images etc.
  + It uses the rest API to interact with the docker demon.

|  |  |
| --- | --- |
| NOTE: The Dockers CLI need not necessarily be on the same host. It can still work with a remote Docker engine. |  |



1. DOCKER DAEMON
   1. The Docker daemon (dockerd) is a background process that runs on the host machine and manages Docker containers.
   2. It listens for Docker API requests and handles container and image operations.
2. DOCKER CLI
   1. The Docker Command Line Interface (CLI) is a command-line tool used to interact with Docker Engine.
   2. It allows us to build, run, and manage containers, images, networks, volumes, and other Docker resources.
3. CONTAINER RUNTIME
   1. Docker Engine uses a container runtime to create and manage containers.
   2. By default, Docker Engine uses the container runtime provided by the operating system, such as runc for Linux-based systems.
4. CONTAINER IMAGES
   1. Docker Engine uses container images as the building blocks for creating containers.
   2. Images contain everything needed to run an application, including the code, runtime, libraries, and dependencies.
   3. Docker Engine uses a layered file system to efficiently store and distribute images.
5. DOCKER REGISTRY
   1. Docker Engine interacts with Docker registries to pull and push container images.
   2. The default public registry is Docker Hub, but we can also use private registries or set up our own registry using Docker Registry or other compatible solutions like Azure Container Registry.
6. NETWORKING
   1. Docker Engine provides networking capabilities that allow containers to communicate with each other and with the external network.
   2. It creates virtual networks, assigns IP addresses to containers, and provides network isolation and port mapping.
7. VOLUMES
   1. Docker Engine supports the use of volumes to persist data generated by containers.
   2. Volumes provide a way to store and share data between containers and the host machine, ensuring data persistence even if a container is stopped or deleted.



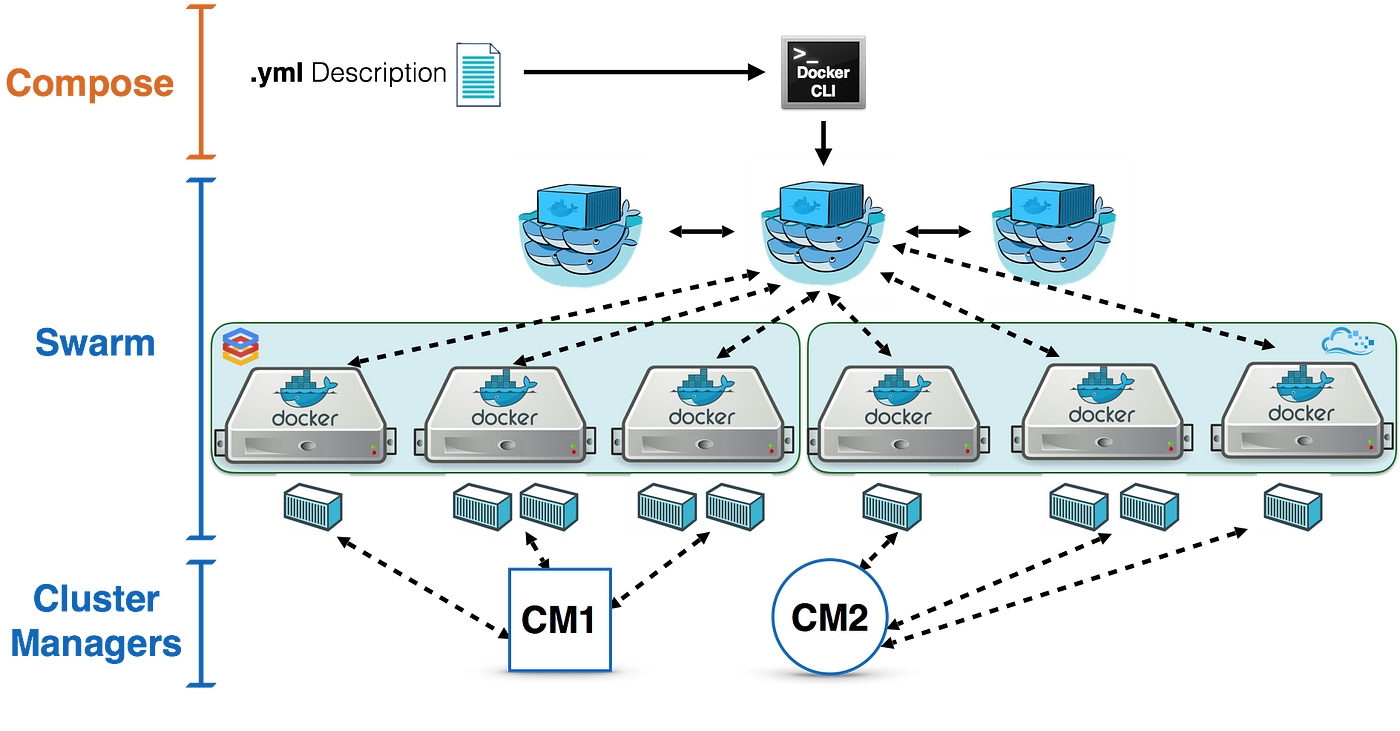
## CONTAINER ORCHESTRATION

* Container orchestration refers to the management and coordination of multiple containers within a distributed environment.
* It involves automating the deployment, scaling, and management of containerized applications across a cluster or fleet of machines.
* Container orchestration systems provide a framework to define, schedule, and manage containerized workloads. They handle tasks such as container placement, scaling, load balancing, service discovery, networking, and resource allocation. These systems ensure that containers are running efficiently, are highly available, and can scale seamlessly as demand fluctuates.  
  Container orchestration platforms typically offer features like:
* SERVICE DEFINITION AND DEPLOYMENT:
  + They provide a way to define the desired state of an application, including the number of containers, resource requirements, dependencies, and configuration parameters. They then deploy and manage the containers accordingly.
* SCALING AND LOAD BALANCING:
  + Container orchestration systems enable automatic scaling of containers based on demand. They distribute the workload across multiple containers and balance the traffic to ensure optimal performance.
* HEALTH MONITORING AND SELF-HEALING:
  + They continuously monitor the health of containers and restart or replace them if they become unhealthy or fail. This ensures high availability and fault tolerance.
* SERVICE DISCOVERY AND NETWORKING:
  + Container orchestration platforms provide mechanisms for containers to discover and communicate with each other. They handle networking configurations, load balancing, and routing traffic to the appropriate containers.
* Resource management:
  + They optimize resource allocation by efficiently utilizing CPU, memory, and storage across the cluster.
  + They also allow for setting resource limits and constraints for individual containers.

Some popular container orchestration platforms include **Docker Swarm, Kubernetes, Apache Mesos, and Amazon ECS (Elastic Container Service)**.

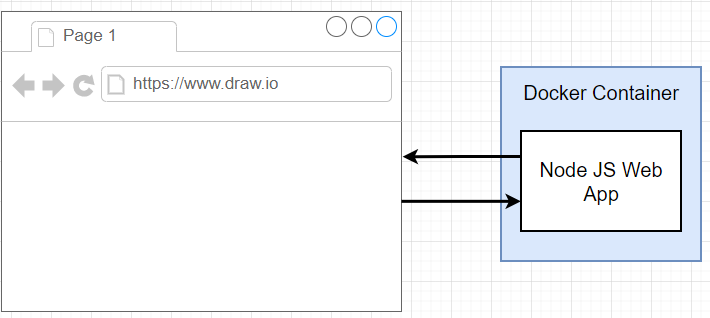
### DOCKER SWARN

* **Docker Swarm is a native clustering and orchestration solution for Docker containers.**



* It allows us to create and manage a swarm of Docker nodes, which are the individual instances of Docker running on different machines.
* With Docker Swarm, we can distribute containerized applications across multiple nodes, providing scalability, fault tolerance, and load balancing.
* **Swarm enables us to create a single virtual Docker host by joining multiple Docker nodes together. This virtual host is called a swarm, and it acts as a single entity to deploy and manage containers.**
* Swarm allows us to define services, which are the desired state of a containerized application, and it ensures that the desired state is maintained across the swarm by automatically managing container placement, scaling, and recovery.
* **USING DOCKER SWARM**
  + We can easily scale our applications horizontally by adding or removing nodes from the swarm.
  + It also provides high availability by automatically rescheduling containers in case of node failures.
  + Swarm supports various scheduling strategies, including spread, binpack, and random, to distribute containers across nodes based on resource availability and constraints.

# A DOCKER PROJECT



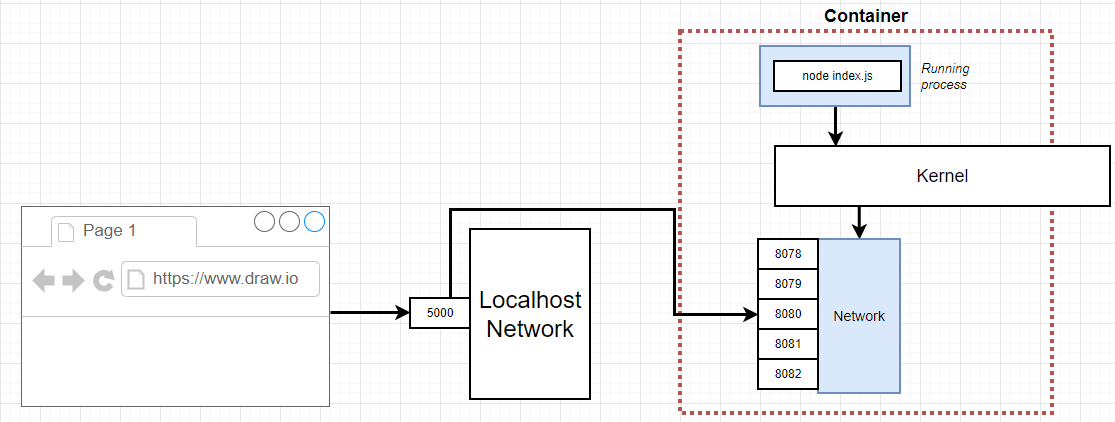
* The goal of the project is to create a tiny Node.js web application 🡪 wrap it inside of a Docker container, and then be able to access that web application from a browser running on your local machine.

## STEPS TO CREATE THE PROJECT

### SETTING UP BASE IMAGE

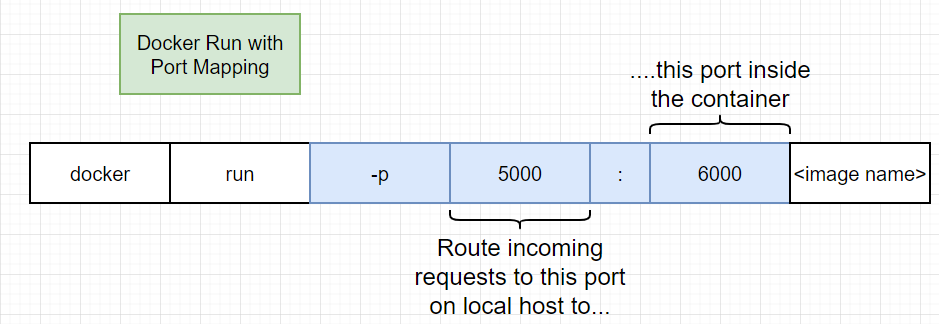
|  |  |
| --- | --- |
|  | * Create a Express App listening on port 8080 with a route , which returns a message * BASE IMAGE : The base image we will be using is node.(as it has npm / node pre-installed in the image) * Tags for node image in Docker Hub : <https://hub.docker.com/_/node> * Dockerfile   **FROM node:14-alpine**  **COPY . /app**  **RUN npm install**  **CMD [ "npm","start" ]**   * TO BUILD THE IMAGE   **docker run -t asinha37/simple-web .** |
|  | |

### PORT MAPPING



* When the browser is making a request to localhost:8080 - By default, no traffic that is coming from local host network is routed into the container. Because the container has its own isolated set of ports that can receive traffic, hence -by default no incoming traffic will be directed into a container.
* To route the traffic to the container we must set up an explicit port mapping. A port mapping essentially says anytime that someone makes a request to a given port on your local network, take that request and automatically forward it to some port inside the container. i.e. if anyone makes a request to localhost:8080, the request will be automatically forward it into the container on port 8080.

#### PORT MAPPING COMMAND



* Ideally – we should do the PORT forwarding in the Docker file .The port forwarding is strictly a runtime constraint. In other words, it's something that we only change when we run a container or start a container up.
* The Application can be accessed using <http://localhost:5000>



### WORKDIR

* In Docker, the WORKDIR instruction is used to **set the working directory for any subsequent instructions** in the Dockerfile. All the further command will be executed from the path specified in WORDIR. It is like the cd command in a Linux/Unix shell.
* The WORKDIR instruction has the following syntax in a Dockerfile: WORKDIR /path/to/directory . Here, /path/to/directory represents the absolute path to the directory we want to set as the working directory inside the container.
* The WORKDIR instruction is typically used to specify the directory where out application code will be copied or where subsequent commands, such as RUN, CMD, or ENTRYPOINT, will be executed. It helps to provide a relative path for subsequent instructions, making the Dockerfile more readable and maintainable.

#### EXAMPLE

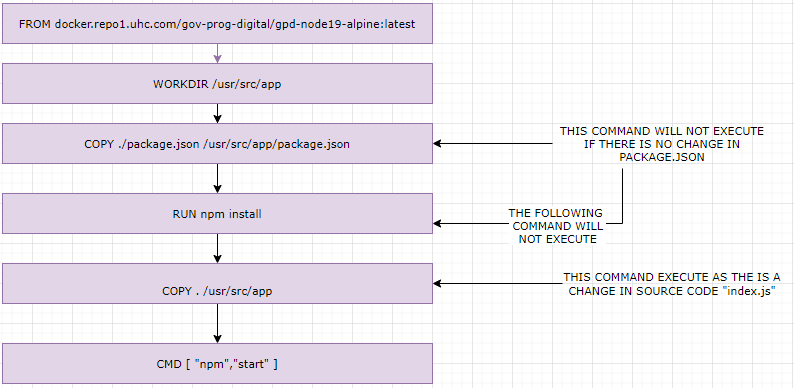
|  |  |
| --- | --- |
| FROM node:14-alpine  **WORKDIR /app**    COPY . .    RUN npm install    CMD [ "npm", "start" ] | * In this example, the WORKDIR /app instruction sets the working directory inside the container to /app. * The subsequent COPY instruction will copy the current directory (where the Dockerfile resides) into the /app directory inside the container. * The RUN instruction will run npm install inside the /app directory, and the CMD instruction will execute npm start within the same directory. |

### MINIMIZING THE CACHE BURSTING AND REBUILDS

* Whenever we make change the source code of the project – Docker detects the changes and run the docker commands (in docker file) accodingly . For example if we updated the code of “index.js” – it runs the command from “COPY” command to get the latest chnages.
* Once the “COPY” command is executed -then it excutes all the following commands
* The Gotcha here is – It runs the “npm install” every time even if we have no change in the dependencies. We want “npm install” to execute only if we have change in “package.json”

|  |  |
| --- | --- |
| DOCKER FILE | PROJECT STRUCTURE |

* **To resolve this issue we have to write our Dockfile in a slighly different way- as shown in below diagram**



# DOCKER COMPOSE- VOTING APP