Contents

[DOCKER 4](#_Toc153568887)

[WHAT IS DOCKER? 4](#_Toc153568888)

[WHY DOCKER? 4](#_Toc153568889)

[USING DOCKER CLIENT? 4](#_Toc153568890)

[WHAT IS CONTAINER? 5](#_Toc153568891)

[RELATION BETWEEN CONTAINER AND IMAGE 6](#_Toc153568892)

[LINUX KERNEL 7](#_Toc153568893)

[DOCKER COMMANDS 7](#_Toc153568894)

[DOCKER RUN 7](#_Toc153568895)

[LIST ALL RUNNING CONTAINER 8](#_Toc153568896)

[STOPING THE RUNNING CONTAINER 8](#_Toc153568897)

[REMOVING THE CONTAINER 9](#_Toc153568898)

[INSPECT CONTAINER 9](#_Toc153568899)

[DOCKER IMAGES COMMAND 9](#_Toc153568900)

[EXECUTING A TASK IN A RUNNING CONTAINER 9](#_Toc153568901)

[EXAMPLES 9](#_Toc153568902)

[ATTACHED AND DETACHED MODE OF RUNNING CONTAINER 10](#_Toc153568903)

[ATTACHED MODE 10](#_Toc153568904)

[DETACHED MODE 10](#_Toc153568905)

[TAGGING 10](#_Toc153568906)

[CONVENTION TO TAG IMAGES 10](#_Toc153568907)

[EXECUTING COMMAND IN A RUNNING CONTAINER 11](#_Toc153568908)

[EXAMPLE 12](#_Toc153568909)

[GETTING TERMINAL ACCESS OF CONTAINER 12](#_Toc153568910)

[CONTAINER - PORT MAPPING 13](#_Toc153568911)

[STEPS TO MAP THE PORT 13](#_Toc153568912)

[EXAMPLE 14](#_Toc153568913)

[DATA PERSISTENCE IN DOCKER CONTAINER 14](#_Toc153568914)

[CONTAINER LOGS 14](#_Toc153568915)

[CREATING A DOCKER IMAGE 15](#_Toc153568916)

[CREATING A DOCKER FILE 15](#_Toc153568917)

[DOCKERFILE IN DETAILS 15](#_Toc153568918)

[DOCKER FILE FORMAT 15](#_Toc153568919)

[DOCKER FILE COMMANDS 16](#_Toc153568920)

[RUNNING THE DOCKER FILE 17](#_Toc153568921)

[LAYERED ARCHITECTURE OF DOCKER BUILD 18](#_Toc153568922)

[ENVIRONMENT VARIABLES IN DOCKER 18](#_Toc153568923)

[COMMAND VERSUS ENTRYPOINT 19](#_Toc153568924)

[DOCKER STORAGE 20](#_Toc153568925)

[DOCKERS LAYERED ARCHITECTURE 20](#_Toc153568926)

[COPY ON WRITE 23](#_Toc153568927)

[DOCKER VOLUME 23](#_Toc153568928)

[DOCKER NETWORKING 25](#_Toc153568929)

[BRIDGE NETWORK 25](#_Toc153568930)

[HOST NETWORK 26](#_Toc153568931)

[USER DEFINED NETWORK 27](#_Toc153568932)

[EMBEDDED DNS 27](#_Toc153568933)

[ASSOCIATING A CONTAINER TO A NETWORK 28](#_Toc153568934)

[DOCKER REGISTRY 28](#_Toc153568935)

[IMAGE NAMES IN REGISTRY 28](#_Toc153568936)

[RUNNING A PRIVATE DOCKER REPOSITORY 28](#_Toc153568937)

[DEPLOYING PRIVATE REGISTRY IN ORGANIZATION 29](#_Toc153568938)

[PUSHING DOCKER IMAGE 29](#_Toc153568939)

[LINKING THE CONTAINER – LEGACY METHOD 30](#_Toc153568940)

[DOCKER COMPOSE 31](#_Toc153568941)

[DOCKER COMPOSE PROJECT 31](#_Toc153568942)

[STEPS TO SET UP DOCKER COMPOSE 32](#_Toc153568943)

[MAINTANING CONTAINER USING DOCKER COMPOSE 33](#_Toc153568944)

[DOCKER ENGINE 34](#_Toc153568945)

[COMPONENTS OF DOCKER ENGINE 34](#_Toc153568946)

[CONTAINER ORCHESTRATION 36](#_Toc153568947)

[DOCKER SWARN 36](#_Toc153568948)

[A DOCKER PROJECT 37](#_Toc153568949)

[STEPS TO CREATE THE PROJECT 37](#_Toc153568950)

[SETTING UP BASE IMAGE 38](#_Toc153568951)

[PORT MAPPING 38](#_Toc153568952)

[WORKDIR 39](#_Toc153568953)

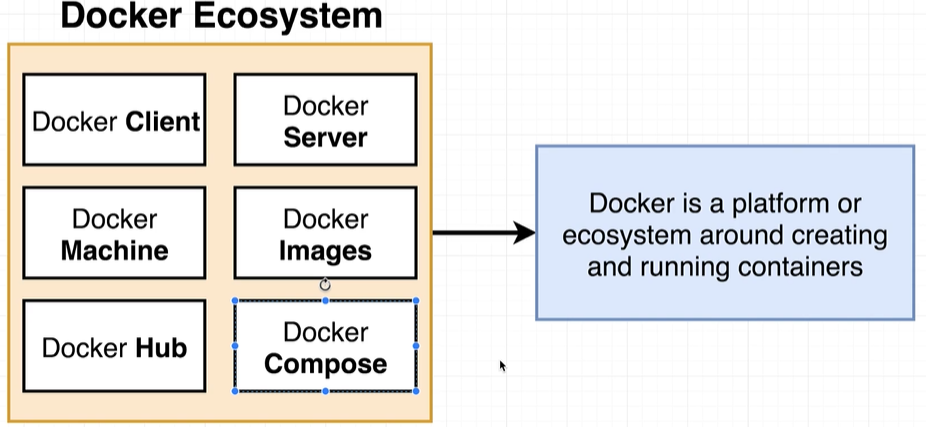
[MINIMIZING THE CACHE BURSTING AND REBUILDS 39](#_Toc153568954)

[PRODUCTION GRADE DOCKER APP 40](#_Toc153568955)

[PROJECT DEPLOYMENT FLOW 41](#_Toc153568956)

# DOCKER

## WHAT IS DOCKER?

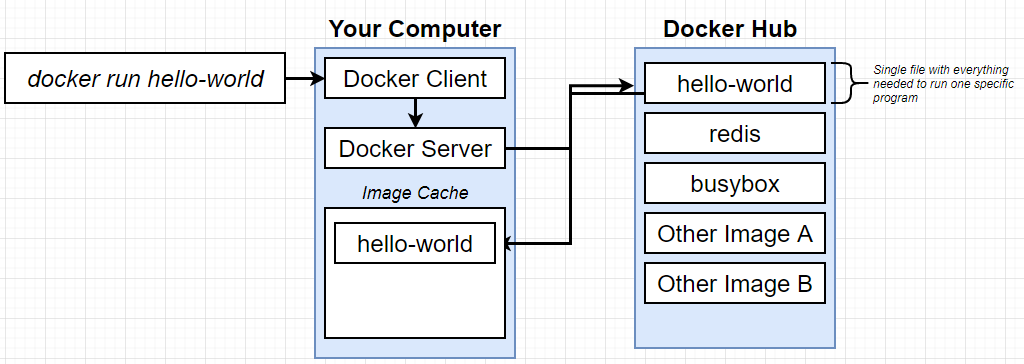


## WHY DOCKER?

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

## USING DOCKER CLIENT?

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



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

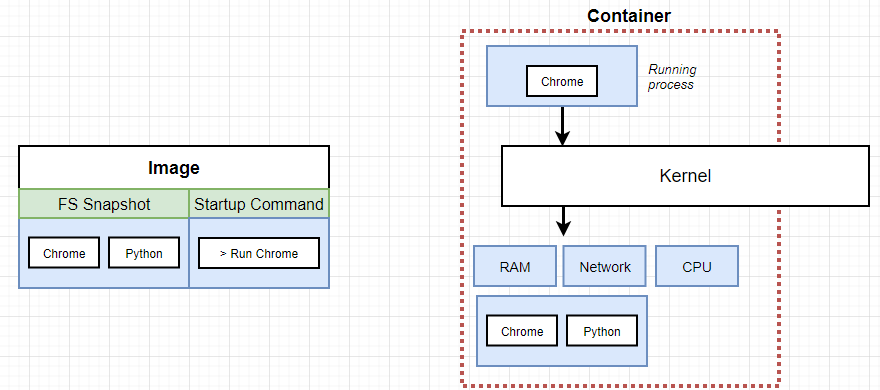
|  |  |
| --- | --- |
| * Once the image is downloaded – It starts a dedicated process called container and start executing the program.   Note: Container is an instance of the image |  |

## WHAT IS CONTAINER?

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

### RELATION BETWEEN CONTAINER AND IMAGE

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



## LINUX KERNEL

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

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

## DOCKER COMMANDS

### DOCKER RUN

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

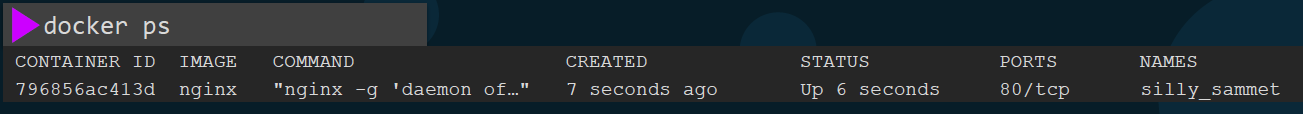
#### OVERRIDING DEFAULT COMMANDS

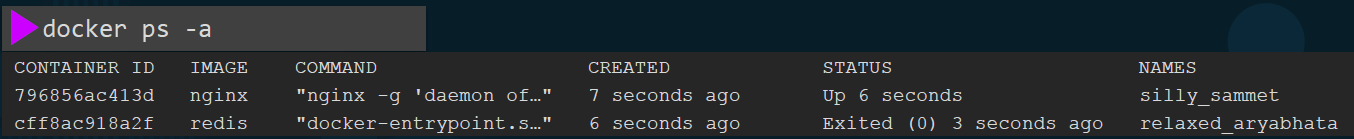
* “docker run” command by default create+ start the container. We can to override the default behavior of these commands. e.g.

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

### LIST ALL RUNNING CONTAINER

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

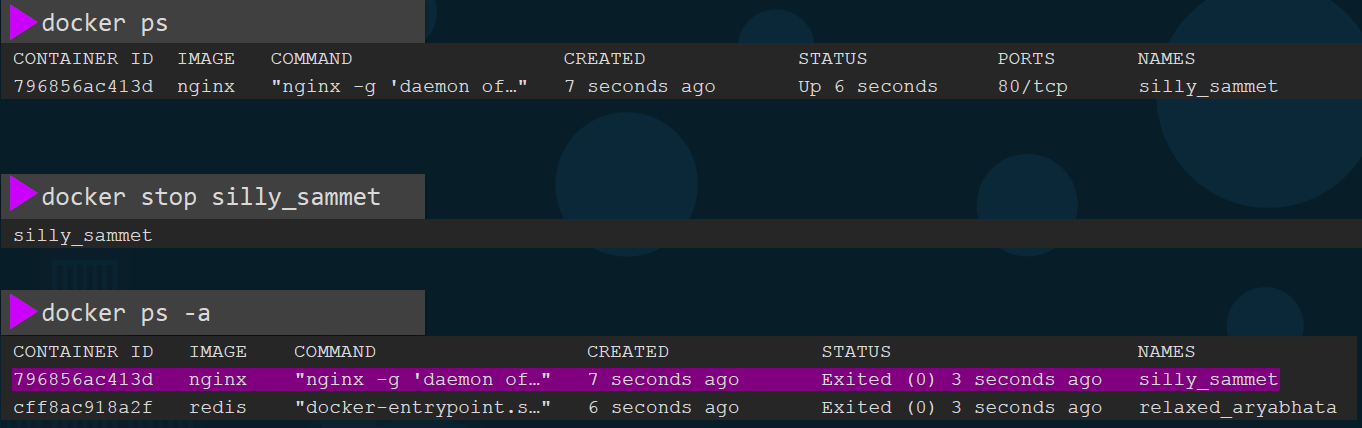




### STOPING THE RUNNING CONTAINER

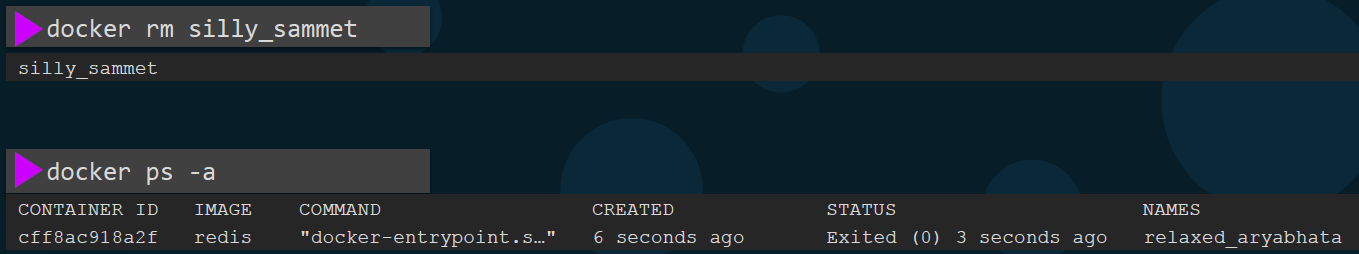
* Stopping the container will change the status from Running 🡪 Exited state.

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



### REMOVING THE CONTAINER

Removes the container

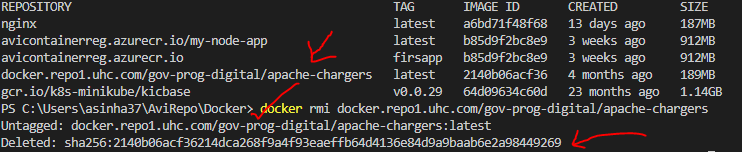


### INSPECT CONTAINER

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

### DOCKER IMAGES COMMAND

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



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

## EXECUTING A TASK IN A RUNNING CONTAINER

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

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

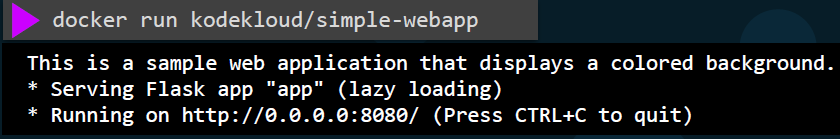
### EXAMPLES

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

## ATTACHED AND DETACHED MODE OF RUNNING CONTAINER

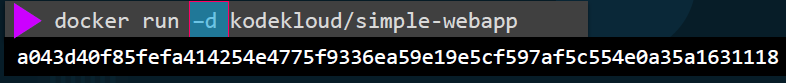
### ATTACHED MODE

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



### DETACHED MODE

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

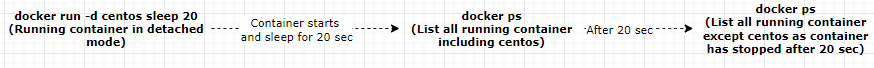


* To attach back to the running container later run the docker attach command and specify the name or I.D. of the docker container.

**NOTE**:



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

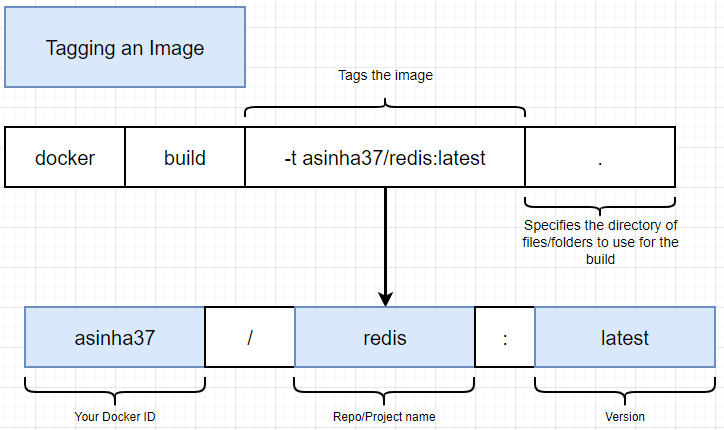


## TAGGING

* In Docker, image tagging is used to assign a specific name and version (tag) to an image.
* Tags are used to differentiate different versions 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**

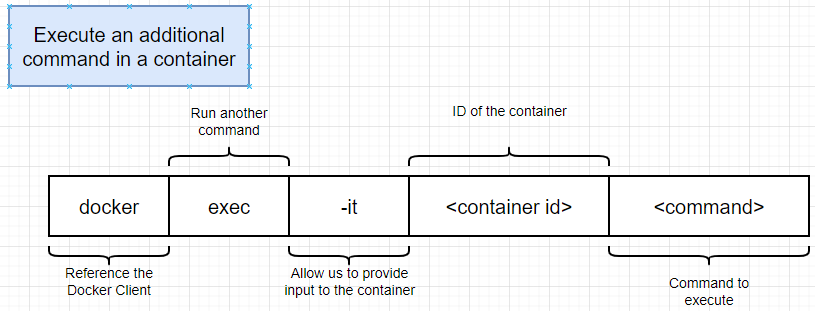


|  |  |
| --- | --- |
| Pulling an image with a specific tag | **docker pull <image\_name>:<tag>** |
| Replace <image\_name> with the name of the image, and <tag> with the desired version or variant of the image | Example- to pull the "nginx" image with the "1.21.1" tag  **docker pull nginx:1.21.1**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

## 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)** |

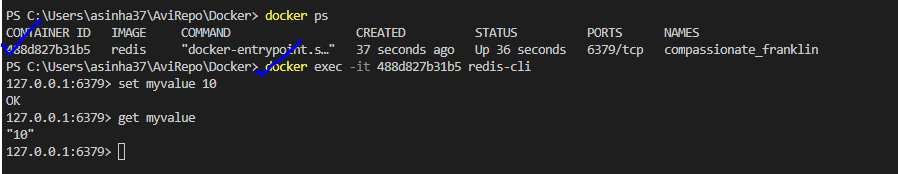


* We can execute commands within the running container using above docker command

**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 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 |

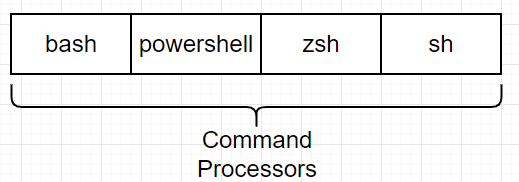
### GETTING TERMINAL ACCESS OF CONTAINER

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

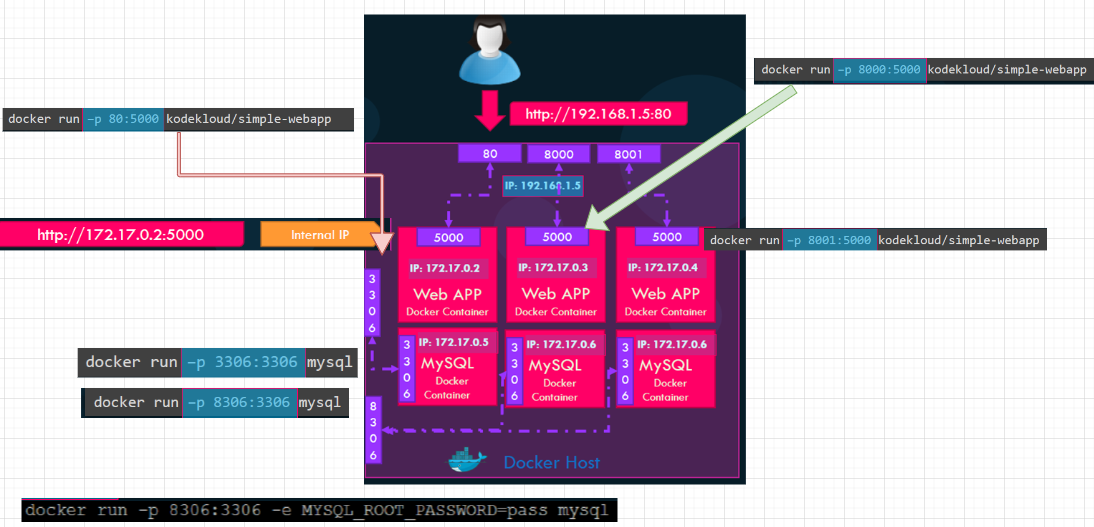
* Port mapping in Docker allows us to map network ports between the host machine and the container.
* This enables us to access services running inside a container from the host machine or other machines on the network.

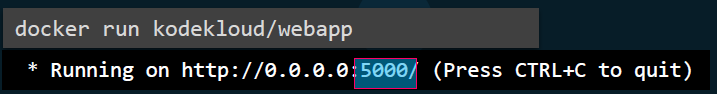
|  |  |
| --- | --- |
| * Use the -p (or --publish) flag with the docker run command * <host\_port>  : The port on the host machine that we want to map. * <container\_port> : The port on the container that we want to expose. | **SYNTAX**  **docker run -p <host\_port>:<container\_port> <image\_name>**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

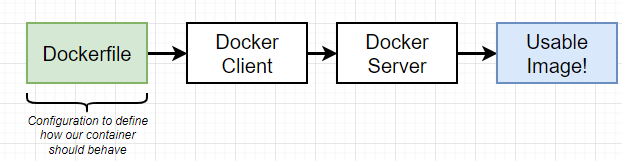
* Volumes can be managed using the docker volume command or specified in a Docker Compose file.

**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

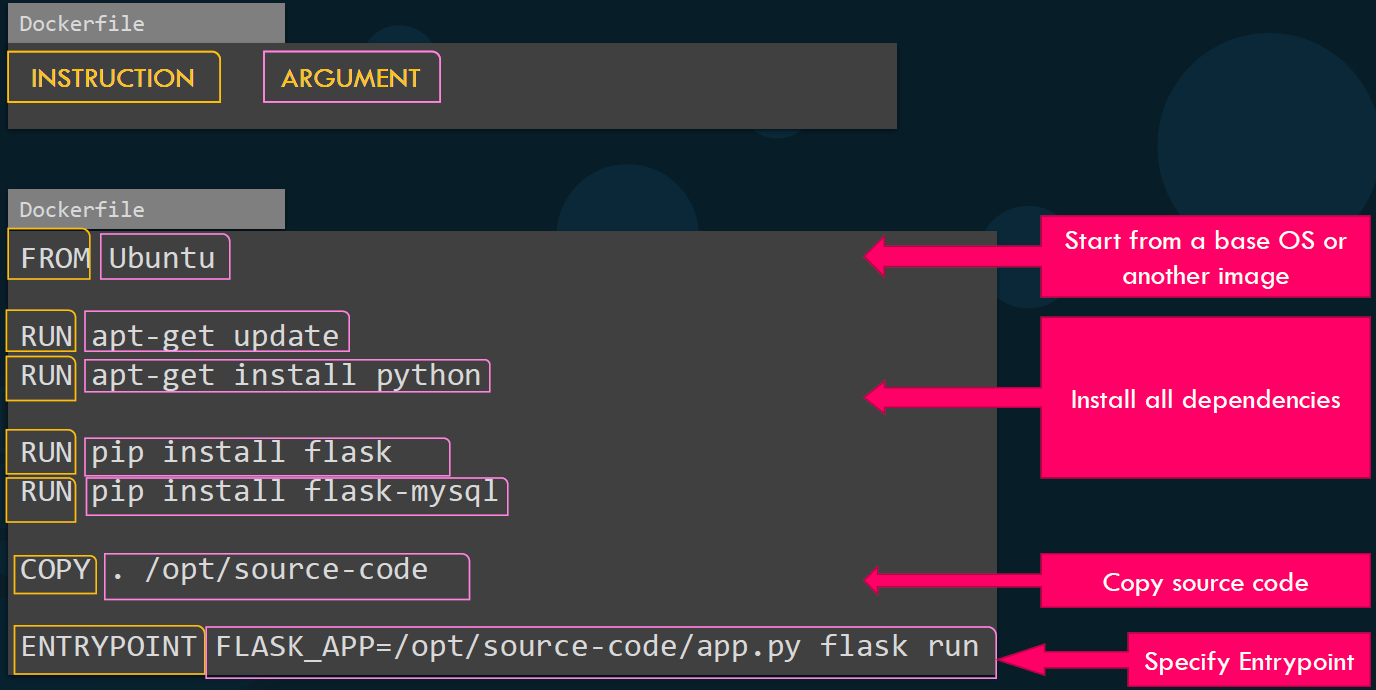
|  |  |
| --- | --- |
|  |  |

### 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

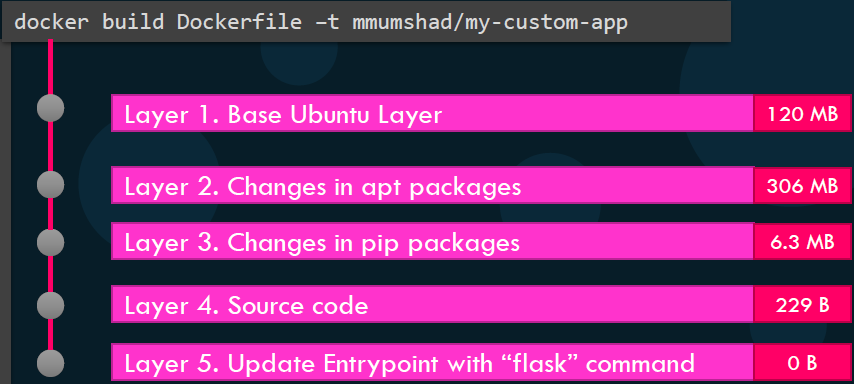
* Once the docker file is created – we can build the docker image

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

### ENVIRONMENT VARIABLES IN DOCKER

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

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

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

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

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

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

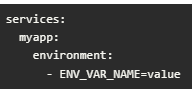
*In your Dockerfile, you can use the ARG instruction to define the build-time argument and then use the ENV instruction to set it as an environment variable.*

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

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

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

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



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

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

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

#### EXAMPLE

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

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

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

### COMMAND VERSUS ENTRYPOINT

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

#### CMD

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

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

#### ENTRYPOINT

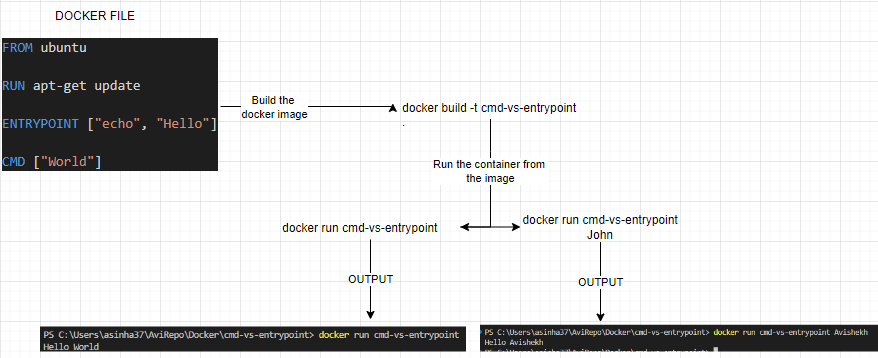
* **The ENTRYPOINT instruction specifies the command that will be executed when the container is started**.
* **ENTRYPOINT is the process that’s executed inside the container.**
* The ENTRYPOINT Dockerfile instruction sets the process that’s executed when the container starts.

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

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

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

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



## 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.

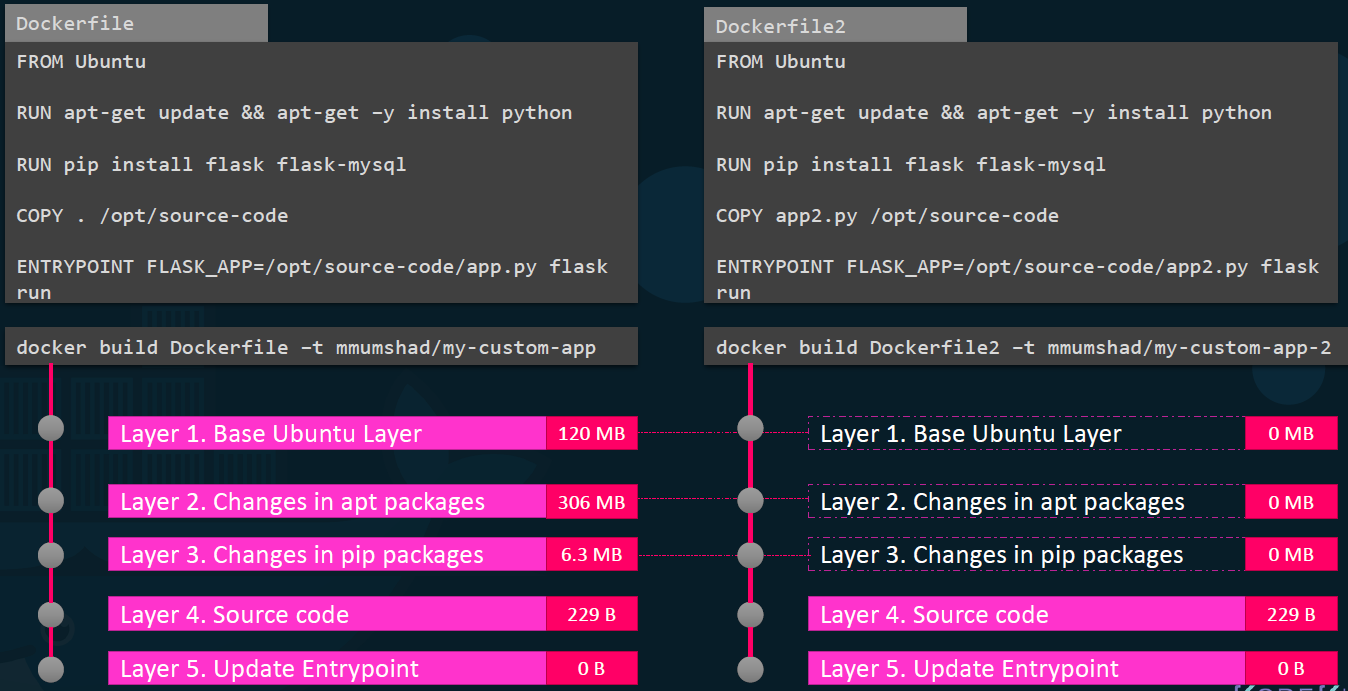
Here's an overview of Docker's layered architecture:

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

* 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.

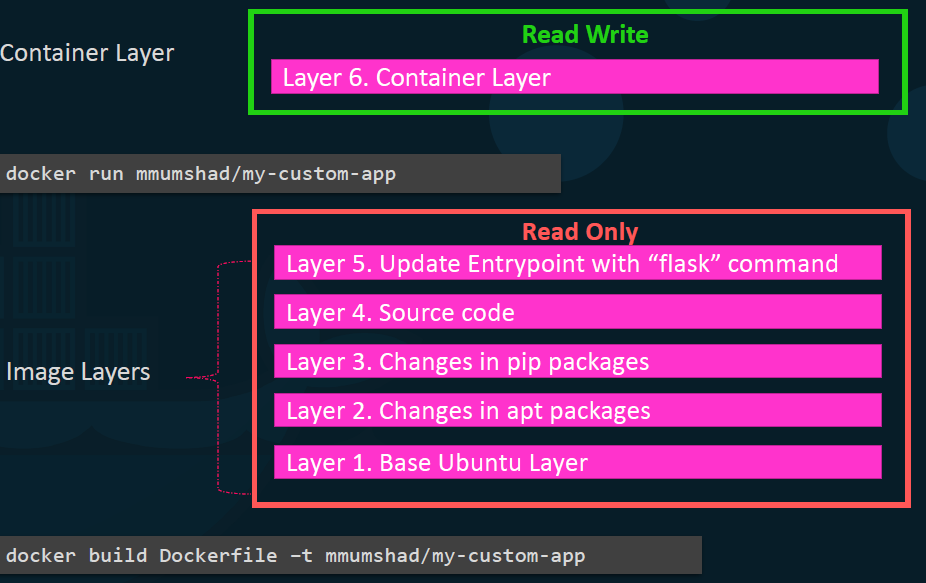


1. The first layer is a base Ubuntu operating system
2. Followed by the second instruction that creates a second layer which installs all the APT packages
3. The third instruction creates a third layer which with the python packages
4. Followed by the fourth layer that copies the source code over.
5. 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

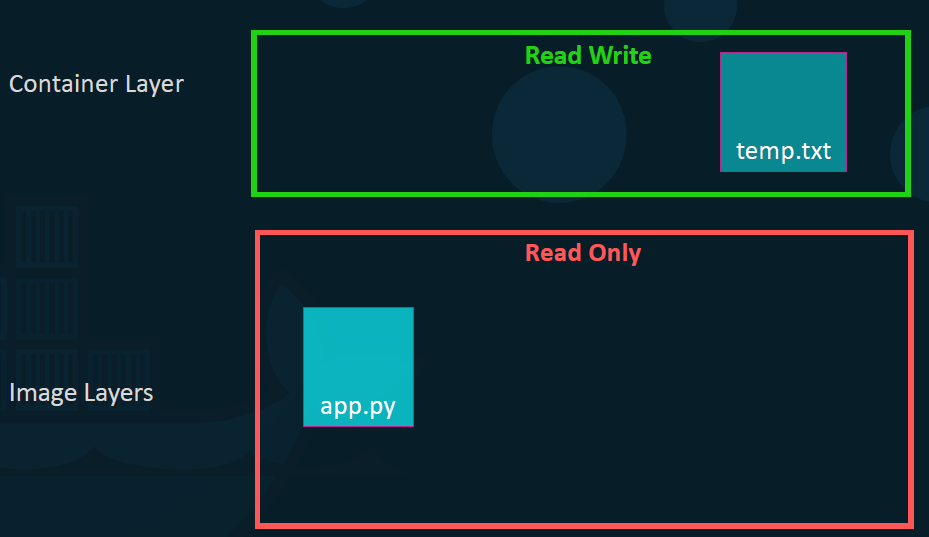
* Bottom we have the base ubuntu layer then the packages then the dependencies and then the source code.

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



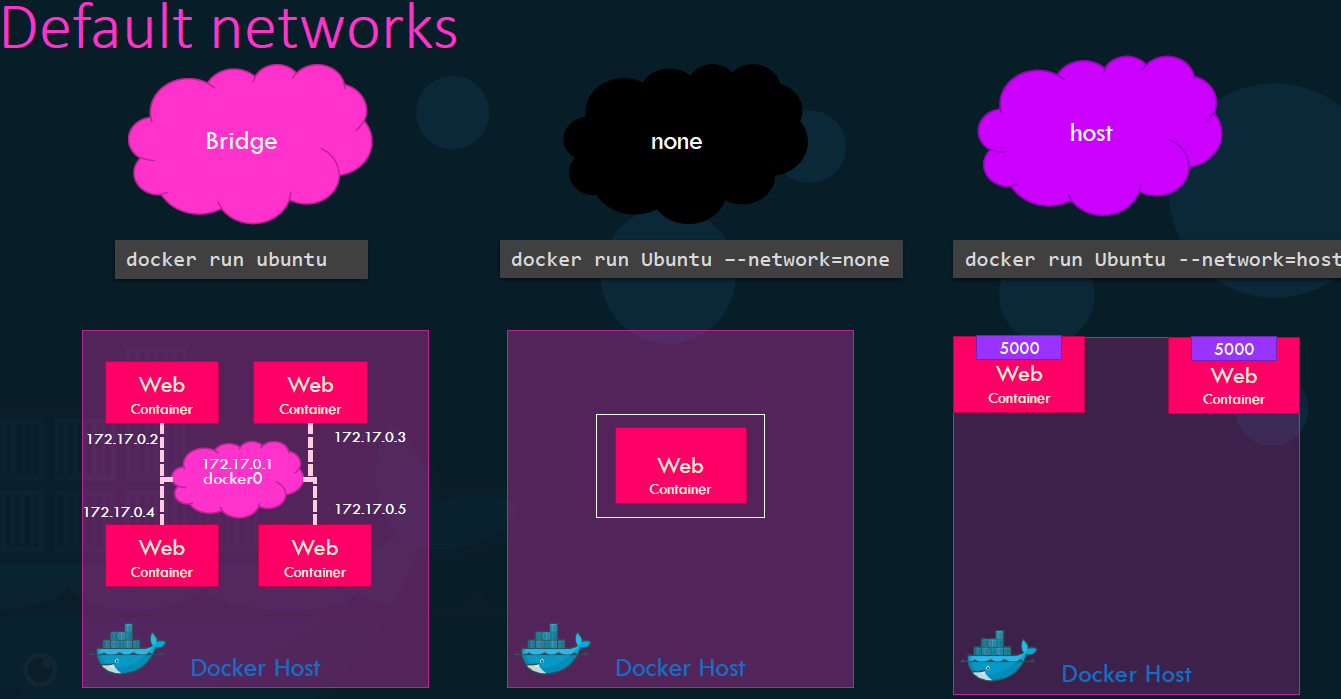
Using the **docker run command** Docker creates a container based off these layers and creates a new writable layer on top of the image layer.

* 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

* **Volumes are directories stored outside the container's file system, allowing data to be shared and reused by multiple containers.**
* **Volumes can be managed by Docker, and they can be attached to specific containers or shared among multiple containers.**
* **Volumes are independent of the container lifecycle, so data stored in volumes persists even if the container is removed or replaced.**
* For example if we were working with a database and we would like to preserve the data created by the
* container we could add a persistent volume to the container to do this first create a volume using the
* docker volume create command.
* So when we run the docker volume create data underscore volume command it creates a folder called data
* underscore volume under the var lib Docker volumes directory.
* Then when I run the docker container using the docker run command I could mount this volume inside the
* docker containers read write layer using the dash v option like this.
* So I would do a docker run -v then specify my newly created volume name followed by a colon and
* the location inside my container which is the default location where MySQL stored data and that
* is var/lib/mysql.
* And then the image name mysql all this will create a new container and mount the data volume we created
* into var/lib/
* Mysql folder inside the container so all data written by the database is in fact stored on the
* volume created on the docker host.
* Even if the container is destroyed the data is still active.
* Now what if you didn't run the docker volume create command to create the volume before the docker run
* command.
* For example if I run the docker run command to create a new instance of mysql container with the
* volume data underscore volume 2 which I have not created yet Docker will automatically create a volume
* named data underscore volume 2 and mount it to the container.
* You should be able to see all these volumes if you list the contents of the var lib Docker volumes folder.
* This is called volume mounting as we are mounting in volume created by Docker under the var lib Docker
* volumes folder.
* But what if we had our data already at another location for example let's say we have some external
* storage on the docker host at /data and we would like to store database data on that volume
* and not in the default var lib docker volumes folder.
* In that case we would run a container using the command Docker run -v
* But in this case we will provide the complete part to the folder we would like to mount.
* That is /data/msql and so it will create a container and mount the folder
* to the container.
* This is called bind mounting.
* So there are two types of mounts a volume mounting and a bind mount volume mount mounts a volume from
* the volumes directory and bind mount mounts a directory from any location on the docker host.
* One final point to note before I let you go using the dash V is an old style the new way is to use dash
* mount option the dash dash mount is the preferred way as it is more verbose.
* So you have to specify each parameter in a key equals value format.
* For example the previous command can be written with the dash mount option as this using the type source
* and target options.
* The type in this case is bind the source is the location on my host and target is the location on my
* container
* so who is responsible for doing all of these operations.
* Maintaining the layered architecture.
* Creating a writable layer moving files across layers to enable copy and write etc. It's the storage drivers.
* So Dockery uses storage drivers to enable layered architecture.
* Some of the common storage drivers are AUFS, BTRFS, ZFS, device-mapper, overlay and overlay 2
* the selection of the storage driver.
* Depends on the underlying OS being used for example with Ubuntu.
* The default storage driver is a ufs whereas this store as driver is not available on other
* operating systems like fedora or cent OS.
* In that case device mapper may be a better option Docker will choose the best stories driver available
* automatically based on the operating system the different storage drivers also provide different performance
* and stability characteristics so you may want to choose one that fits the needs of your application
* and your organisation.
* If you would like to read more on any of these storage drivers please refer to the links in the attached
* documentation for now.
* That is all from the docker architecture concepts.
* See you in the next lecture.

## DOCKER NETWORKING



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

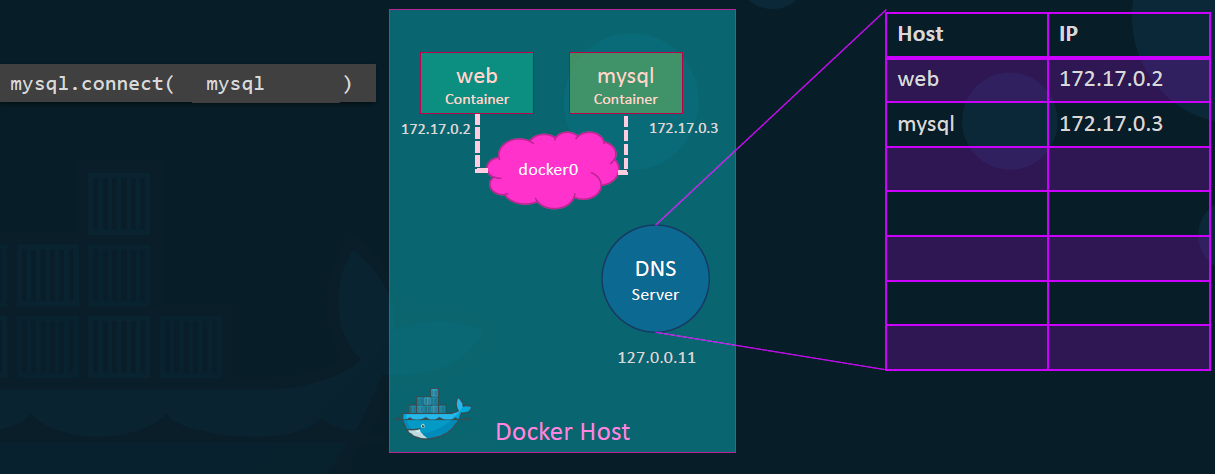
1. **BRIDGE** 
   1. A bridge network is a default network that allows communication between containers on the same Docker host.
   2. When we create a new container without specifying a network, it is automatically connected to the bridge network.
2. **NONE**
3. **HOST BRIDGE**

### 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.

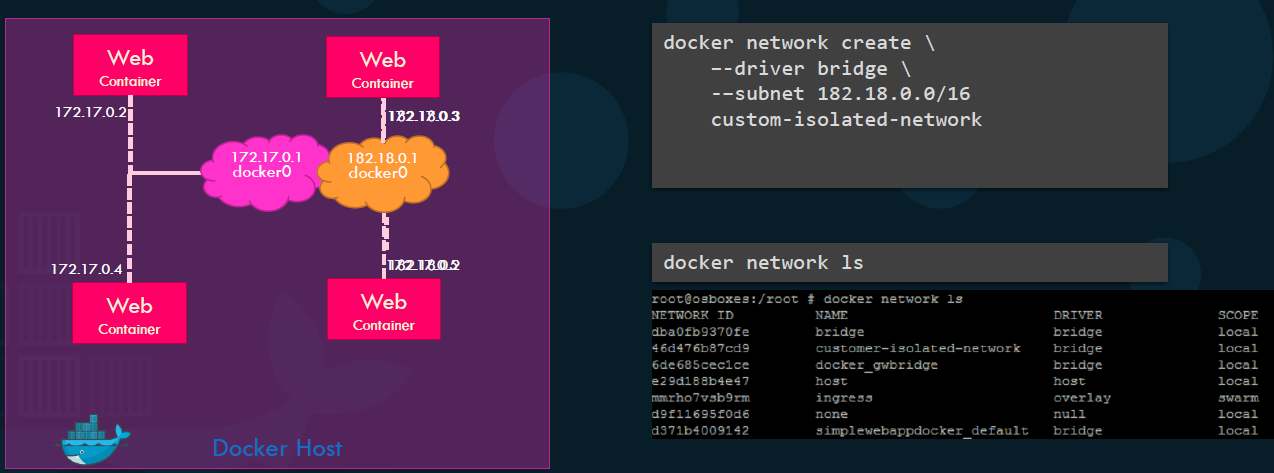
### HOST NETWORK

* **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*

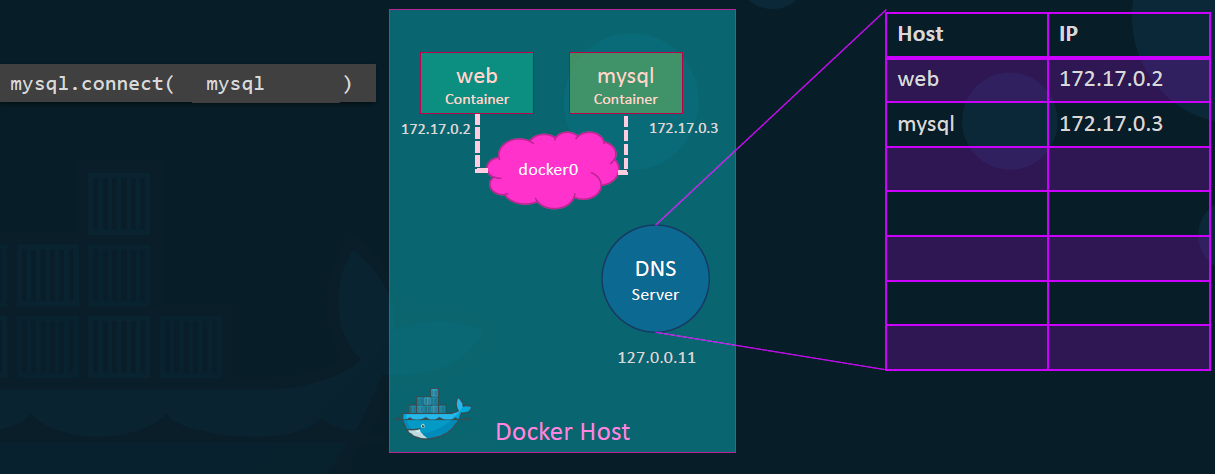
**docker network create [OPTIONS] NETWORK\_NAME**

**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.

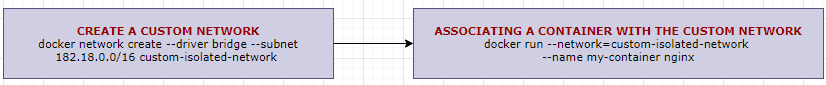
#### HOW EMBEDDED DNS WORKS IN DOCKER?

* **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 |



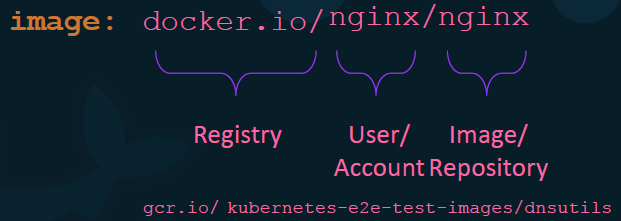
## DOCKER REGISTRY

* 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

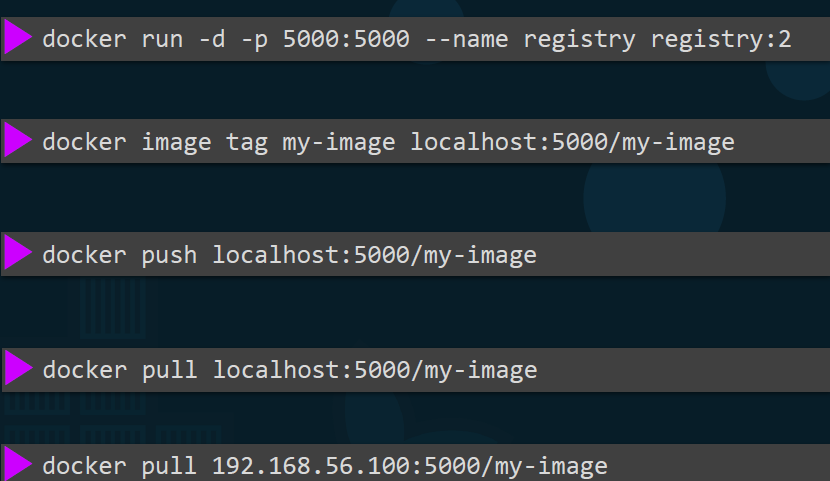
### IMAGE NAMES IN REGISTRY

****

### 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**

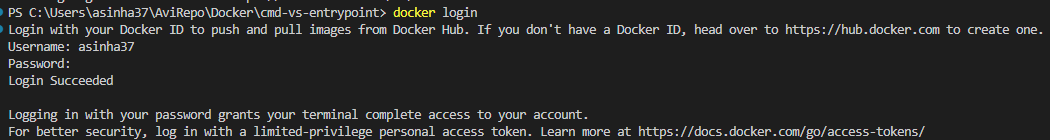
Now its time to push some images to our registry server. Let's push two images for now .i.e. nginx:latest and httpd:latest.

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.



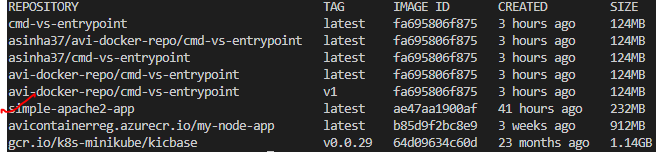
1. **TAG IMAGE**:
   1. Before pushing the image, ensure that it is properly tagged with your Docker Hub username and the repository name.
   2. Use the docker tag command to tag your image.

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

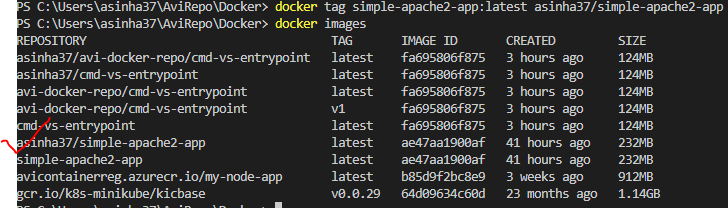
*Replace your-docker-username with your Docker Hub username and your-repository-name with the desired repository name*

**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**

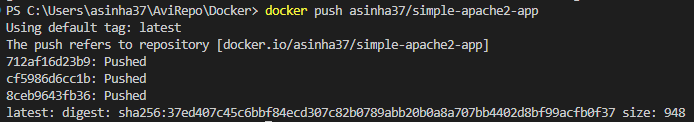


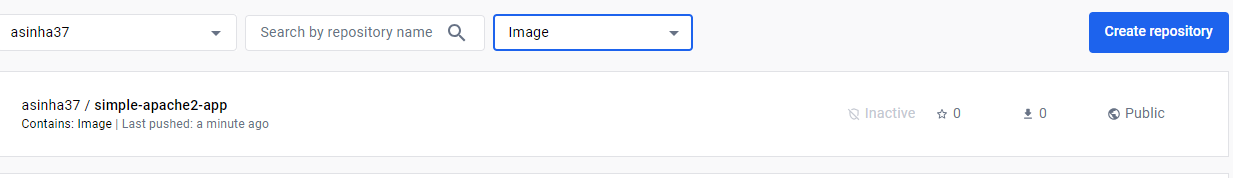
1. PUSH THE IMAGE: Use the docker push command to push the tagged image to Docker Hub.

**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.

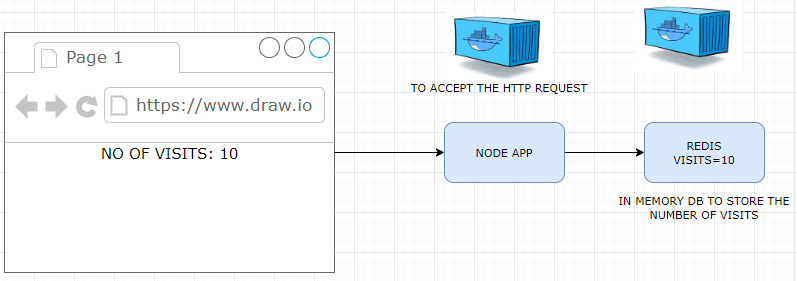
You can use these environment variables in your application code to connect to the linked container's network services.  
  
Please note that the --link flag is considered a legacy feature in Docker, and it's recommended to use user-defined networks for container communication instead. User-defined networks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

## DOCKER COMPOSE

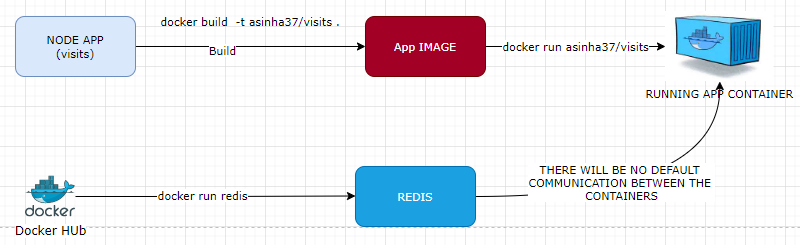
* Docker Compose is a tool that allows us to define and manage multi-container Docker applications.
* It is used to simplify the process of running multiple interconnected containers as a single application stack.
* With Docker Compose, we can define a YAML file called **docker-compose.yml** that contains 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.

### 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.

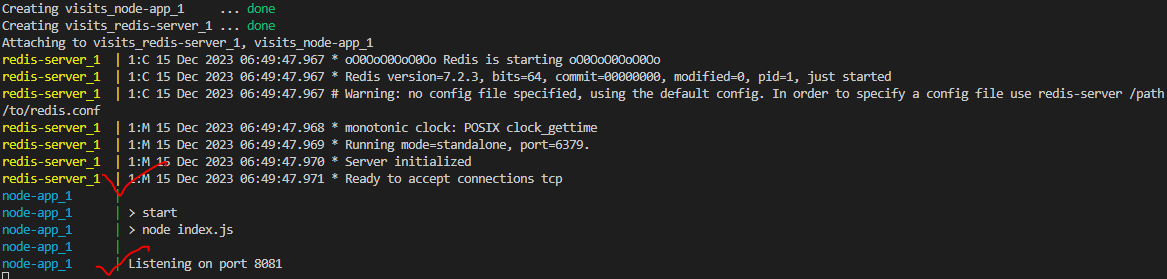
|  |  |
| --- | --- |
| **version: '3'**  **services:**  **redis-server:**  **image: 'redis'**  **node-app:**  **build: .**  **ports:**  **- "4001:8081"** | * We have configured 2 services “redis-server” and “node-app” * node-app is a custom image   + build : Path of docker image   + ports: Port mapping [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. |

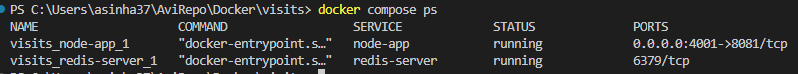
* **The below logs show that the docker compose set up a network to the communication between the containers**





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**

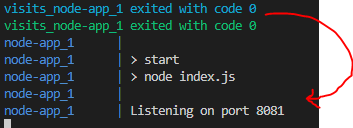
|  |  |
| --- | --- |
| **services:**  **my-service:**  **restart: always** | * We can also set the restart policy in a Docker Compose file using the restart key under the services section. * This sets the restart policy for the my-service container to "always" in the Docker Compose file. |

##### 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>

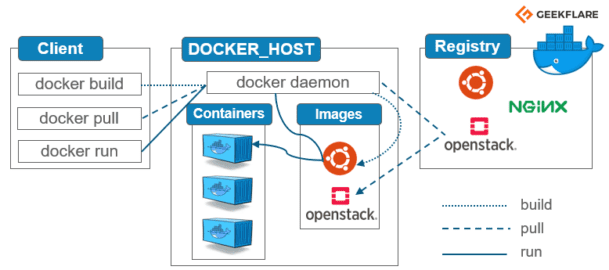
## DOCKER ENGINE

* 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.

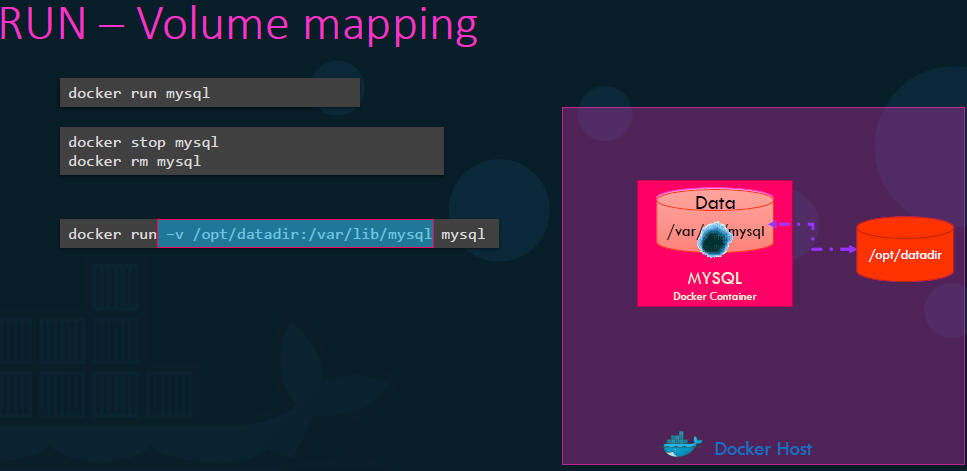
### COMPONENTS OF DOCKER ENGINE

* 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.



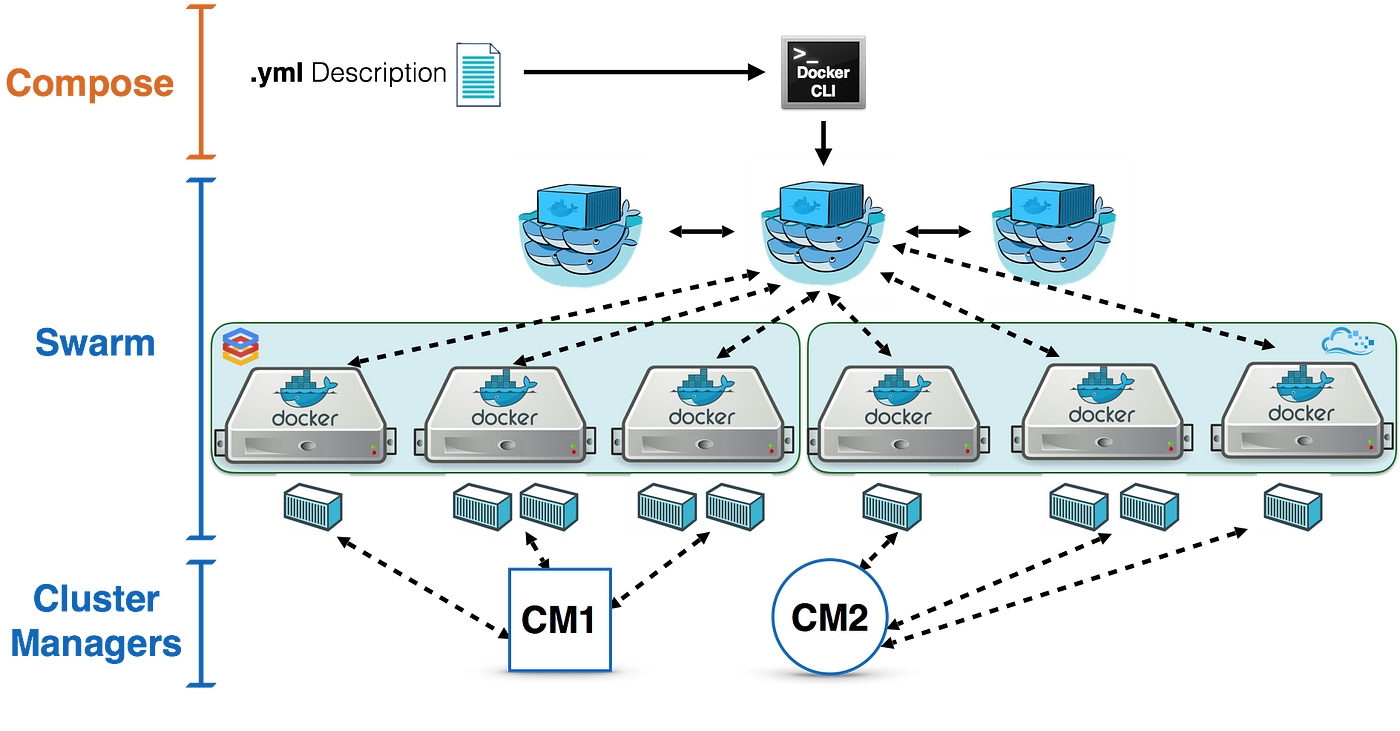
## CONTAINER ORCHESTRATION

* 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)**.

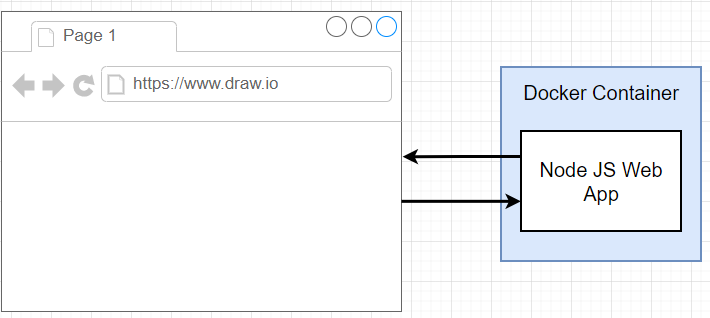
### DOCKER SWARN

* **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



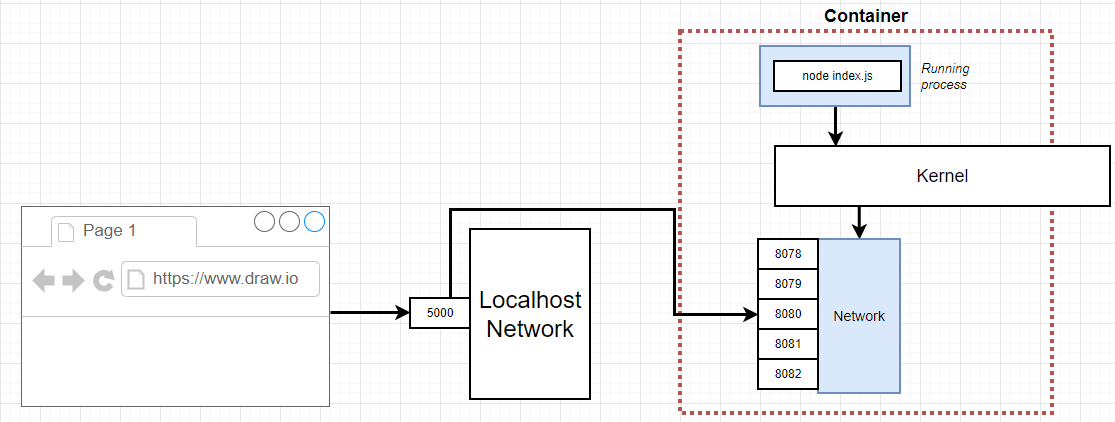
* The goal of the project is to create a tiny Node.js web application 🡪 wrap it inside of a Docker container, and then be able to access that web application from a browser running on your local machine.

## STEPS TO CREATE THE PROJECT

### SETTING UP BASE IMAGE

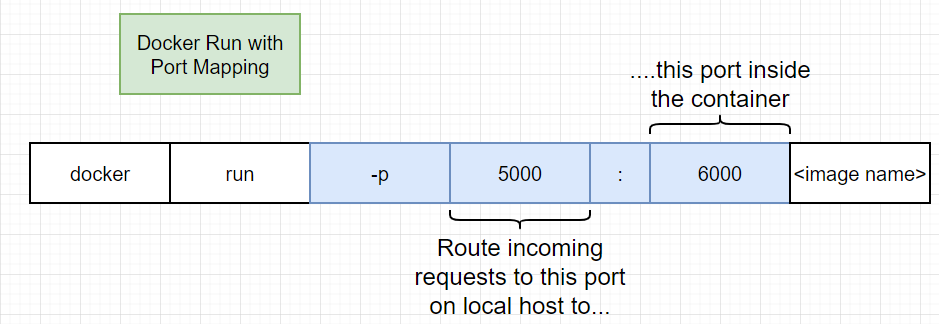
|  |  |
| --- | --- |
|  | * Create a Express App listening on port 8080 with a route , which returns a message * BASE IMAGE : The base image we will be using is node.(as it has npm / node pre-installed in the image) * Tags for node image in Docker Hub : <https://hub.docker.com/_/node> * Dockerfile   **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

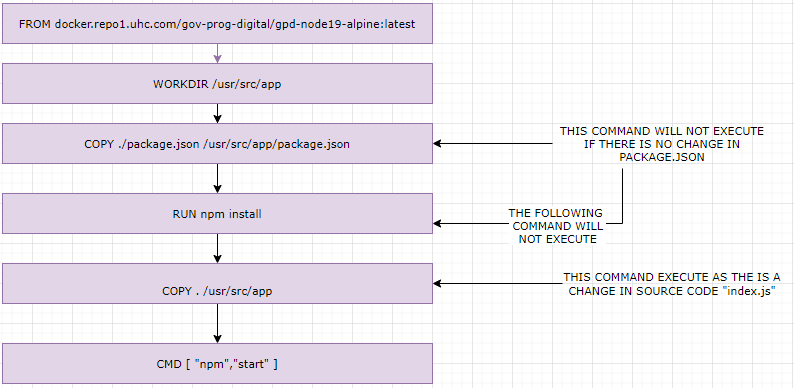
|  |  |
| --- | --- |
| FROM node:14-alpine  **WORKDIR /app**    COPY . .    RUN npm install    CMD [ "npm", "start" ] | * In this example, the WORKDIR /app instruction sets the working directory inside the container to /app. * The subsequent COPY instruction will copy the current directory (where the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