**Docker**

**&**

**Kubernetes**

Table of Contents

[Docker 3](#_Toc179186839)

[Container 3](#_Toc179186840)

[Why would we want **independent, standardized “application packages”?** 3](#_Toc179186841)

[Virtual Machines vs Docker Containers 3](#_Toc179186842)

[VM 3](#_Toc179186843)

[Docker 3](#_Toc179186844)

[Docker Tools & Building Blocks 4](#_Toc179186845)

[Docker Engine 4](#_Toc179186846)

[Docker Desktop 4](#_Toc179186847)

[Docker Hub 4](#_Toc179186848)

[Docker Compose 4](#_Toc179186849)

[Dockerfile 4](#_Toc179186850)

[Images and Containers 5](#_Toc179186851)

[Images are Read-Only! 5](#_Toc179186852)

[Layer Architecture 6](#_Toc179186853)

[Summary 7](#_Toc179186854)

[Managing Images and Containers 7](#_Toc179186855)

[Images 7](#_Toc179186856)

[Containers 7](#_Toc179186857)

[Understanding Attached and Detached Containers 7](#_Toc179186858)

[Deleting Images & Containers 7](#_Toc179186859)

[Copying Files into & from a container 7](#_Toc179186860)

[Naming & Tagging Containers and Images 8](#_Toc179186861)

[Sharing Images 8](#_Toc179186862)

[Managing Data and Working with Volumes 9](#_Toc179186863)

[Volumes 9](#_Toc179186864)

[Two Types of External Data Storages 9](#_Toc179186865)

[Volumes (Managed by Docker) 9](#_Toc179186866)

[Anonymous Volumes 9](#_Toc179186867)

[Named Volumes 10](#_Toc179186868)

[Bind Mounts (Managed by us) 10](#_Toc179186869)

[Understanding Container / Volume Interaction 11](#_Toc179186870)

[Managing Docker Volumes 12](#_Toc179186871)

[COPY vs Bind Mount 13](#_Toc179186872)

[.dockerignore file 13](#_Toc179186873)

[ARGuments and ENVironment Variables 13](#_Toc179186874)

[Networking: (Cross-) Container Communication 14](#_Toc179186875)

[Docker Networks 14](#_Toc179186876)

[Docker Compose 16](#_Toc179186877)

[docker-compose.yaml 16](#_Toc179186878)

[Commands 17](#_Toc179186879)

[Questions 21](#_Toc179186880)

[Best Practice 23](#_Toc179186881)

# Docker

* Is a container technology.
* A tool for creating and managing containers.
* It can be installed on all modern OS.

## Container

* A standardized unit of software.
* A package of code and dependencies to run that code (Ex – NodeJS code + the NodeJS runtime)
* The same container always yields the exact same application and execution behavior! No matter where or by whom it might be executed.
* Support for containers is built into modern operating systems.

## Why would we want **independent, standardized “application packages”?**

* Different Development & Production Environments.
* Different Development Environments within a Team/Company.
* Clashing Tools / Versions Between Different Projects.

We want Reliability and Reproducible Environments.

* We want to have the exact same environment for development and production -> this ensures that it works exactly as tested.
* It should be easy to share a common development environment / setup with (new) employees and colleagues.
* We don’t want to uninstall and re-install local dependencies and runtimes all the time.

## Virtual Machines vs Docker Containers

### VM

* Every virtual machine is really like a standalone computer, a standalone machine running on top of our machine and therefore, if we have multiple such machines especially, we have a lot of wasted space and resources because every time a brand-new computer must be setup inside our machine and that, of course eats up memory, CPU and space on our hard drive.
* Pro
  1. Separated environments
  2. Environment specific configurations are possible
  3. Environment configurations can be shared and reproduced reliably
* Cons
  1. Redundant duplication, waste of space
  2. Performance can be slow, boot times can be long
  3. Reproducing on another computer / server is possible but may still be tricky.

### Docker

* With Containers we still have our host OS, we utilize built-in container support which our OS has or emulated container support, something that Docker takes care of.
* Then we run a tool called the Docker Engine on top of that which comes with Docker installation.
* And these spin up multiple containers which contain code, crucial tools and runtimes respectively
* They don’t bloat the OS
* They might contain a lightweight OS layer.
* You can configure and describe them with a configuration file
* Share that file with others to recreate it
* Or We can build the container into an image and that you can share that image to launch the exact application.
* Docker
  1. Low impact OS, very fast minimal disk space usage
  2. Sharing, re-building and distribution is easy
  3. Encapsulate apps / environments instead of whole machines
* VM
  1. Bigger impact on OS, slower, higher disk space usage.
  2. Sharing, re-building and distribution can be challenging.
  3. Encapsulate “whole machine” instead of just apps/ environments.

## Docker Tools & Building Blocks

### Docker Engine

### Docker Desktop

* It includes Daemon, a process which keeps on running and ensures that Dockers works or the heart of Docker.
* It contains a CLI which helps to run commands, to create images and containers.

### Docker Hub

* This will be a service which will allow us to host our images in the cloud / web to be shared.

### Docker Compose

* This tool makes managing more complex containers or multi-container projects easier.

## Dockerfile

Example 1

# We want to use Node.js as a base image, so that we want to have Node.js available inside of our container

FROM node:20

# Certain directory in the container file system, every container has its own file system.

# We want to have a special directory in there in which we want to work.

WORKDIR /app

# copy the package.json file into our working directory.

COPY package.json .

# we Run the npm install command to install dependencies

RUN npm install

# we copy the rest of the code

COPY . .

# We expose the port 3000 to the outside world and as the application run on that port

EXPOSE 3000

# then we execute app.mjs with node command

CMD ["node", "app.mjs"]

Example 2

FROM node:20

# Sets the working directory to execute the commands below this

WORKDIR /app

# COPY . .

# the first . signifies the path of host file system

# the second . signifies inside the image/ container file system or the root folder system

# /app will be created inside image

COPY . /app

# ./ points to /app also

RUN npm install

# the above commands to set the image

EXPOSE 80

# To run the image in the container

CMD ["node", "server.js"]

# Images and Containers

* Blueprints / templates for containers which then are running instances with read and write access.
* Contains the code and required tools / runtime to execute.
* We create an image with all the setup instructions and all our code once and then use the image to create multiple containers based on that image to run it on multiple timed on different machines, different servers.
* Image is that Shareable package with all the setup instructions and all the code.
* The containers will be the concrete running instance off such an image.
* Images are the blueprints and the templates which contained the code and the application, and containers are then the running application.

A diagram of a software application

Description automatically generated

* Two ways of creating or getting an image so that we can run the container.
  1. Use an existing, pre-built image **Source** – Docker Hub

Ex – Node Official Image

Command – **docker run node**

* + - This command here will use the node image which we will find on Docker Hub, and it will utilize it to create a so-called container based on this image, in this case the image contains the node installation.
  1. Create your own, custom image – Write your own Dockerfile (based on another image)

## Images are Read-Only!

* Whenever we run the docker build command we get a locked and finished image, Images are readable only. So, if we change anything in the code after the build, we need to build again.
* Every time we build, a new image is created if there is any change observed.
* Images are closed template in the end once a build is done.
* If there is no change and we build again, it uses the cache to understand and gives the same build.

## Layer Architecture

* When we rebuild, only the instruction where something changed, and all the instructions thereafter are re-evaluated, and a new build is created.
* Docker basically recognizes that all the instructions, where there is no change will be the same as before.
* When we build Docker caches all the instructions and when we rebuild utilizes the cache to the instructions where there is no change is noticed and re-evaluates the commands thereafter.
* This is called as Layer based architecture where every instruction is a Layer.
* Image is built up from multiple Layers based on these different instructions.
* Every instruction creates a layer, and the layers are cached and if we run a container based on an image, it creates a new extra layer on top of the image which is running the application i.e., CMD instruction.
* The caching helps in speeding up the image creation process and re-executes what needs to be re-executed.

A screenshot of a computer

Description automatically generated

* The way of using layer makes the way for optimization in the Example 2 as below
* FROM node:20
* # Sets the working directory to execute the commands below this
* WORKDIR /app
* # this instructions way is optimized as the dependencies mentioned in package.json doesn’t change frequently than code,
* # so the below step can be executed from cache
* COPY package.json /app
* RUN npm install
* # the above commands to set the image
* # COPY . .
* # the first . signifies the path of host file system
* # the second . signifies inside the image/ container file system or the root folder system
* # /app will be created inside image
* COPY . /app
* # ./ points to /app also
* EXPOSE 80
* # To run the image in the container
* CMD ["node", "server.js"]

### Summary

* Container doesn’t contain the environment, it will use the environment stored in an image, and then just add this extra layer on top of it and allocate resources, memory and so on to run the application.
* If we have multiple containers running on the same code, our code and environment is not getting copied multiple times. It exists only once in the image and to containers, then utilize that image and the code in it. This is how Docker manages efficiently.

## Managing Images and Containers

## Images

* Can be tagged (named) **-t, docker tag** …
* Can be listed **docker images**
* Can be analyzed **docker image inspect**
* Can be removed **docker rmi, docker prune**

## Containers

* Can be named **–name**
* Can be configured in detail see **–help**
* Can be listed **docker ps**
* Can be removed **docker rm**

## Understanding Attached and Detached Containers

* docker start <container\_name>
* This command starts the stopped container and run in the background unblocking terminal for other process and called Detached mode.
* docker run ps -p 8000:80 <build\_id>
* This blocks the terminal, and every time starts in a new container and called attached mode where we listen what happens in that container.
* To run the container in detached mode – docker run -p 8000:80 **-d** <build\_id>
  1. -d represents for detached mode.
  2. Executing this code gives the container id, if you wanted to attach to it again you can with that id.
* **docker attach <container\_name or container\_id>** to attach again or by **docker logs -f <container\_name> or**
* **docker start -a -i <container\_name> -** To start the container in attached and interactive mode
* **docker run -it <build\_id>** - To run the container in interactive mode

## Deleting Images & Containers

* To delete the container, the container needs to be stopped first
* **docker rm <container\_id>**
* To delete the image, the container should be stopped and deleted
* **docker rmi <image\_id>**
* To delete all the images not being used at once – **docker image prune**
* Automatically remove the container when it exits – **docker run -p 3000:80 -d –rm <image\_id>**

## Copying Files into & from a container

* To copy file into or out of a running container –

From outside to inside container

**docker cp <source\_folder\_path> <container\_name/container\_path>**

From inside of container to outside

**docker cp <container\_name/container\_path> <source\_folder\_path>**

* This command can be used for couple of things
  1. It allows you to add something to a container without restarting the container and rebuilding the image like configuration files.
  2. You can use cp command all the log files from inside to outside.

## Naming & Tagging Containers and Images

* Naming a container

docker run -p 3000:80 -d –rm –name <user\_provided\_container\_name> <image\_id>

or   
docker run -p 3000:80 -d –rm –name <user\_provided\_container\_name> <image\_name>:<tags>

* --name helps us to give the name
* Assigning name to images, that called Tag, which has two parts
* **name: tag**
  1. name – repository of our image, or general name,
     + defines a group pf possible more specialized name, images

Ex - node

* 1. tag – defines a specialized image within a group of images.
     + Example – 14
* If an image has no tag, the name **alone** is also a unique identifier.
* -t : Name and optionally a tag in the 'name:tag' format
* **docker build -t goals:latest .**
* Renaming a Tag

**docker tag <old\_name>:<tag> <new\_name>:<new\_tag>**

* When you rename the image, it creates a clone of the same image with the new name

## Sharing Images

* There are two ways of sharing
  1. Sharing the Docker file and the source code, which can be used to build an image and run the container.
  2. Share a built Image – Download an image, run a container based on it.
     + No build step required, everything is included in the image, already!
* There are two main places where we can push our images
  1. Docker Hub – Official Docker image registry
  2. Private Registry
* To share the Image to any registry – **docker push IMAGE\_NAME**
* To pull the Image from the registry – **docker pull IMAGE\_NAME**
* By default, if no HOST:NAME is mentioned it goes to Docker Hub, if Host is mentioned along with image name, it goes to the private registry.
* docker pull always fetches the latest available image.
* We can also use the docker run command to run application, If the image is not available in our system, it checks globally and fetches the latest from global Docker Hub and runs the image, but if the image is available in the local, it run that irrespective of the version available.

# Managing Data and Working with Volumes

* Apart from the Source code, we have two other Data
  1. Temporary App Data –
     + Ex- Data entered by user in a Form, which is stored in memory or temporary files.
     + Dynamic and changing but cleared regularly
     + Since it is read + write it is stored in containers
  2. Permanent App Data –
     + Ex – User accounts
     + Fetched / Produced in running container
     + Stored in files or a database
     + Must not be lost if container stops / restarts
     + Read + write permanent, stored with Containers and Volumes
* The downside of keeping the files on containers, once the container is removed or deleted the files are lost.
* It doesn’t affect if the container is stopped and on starting back the files are available.

## Volumes

* Help us with persisting data.
* Are folders on your host machine hard drive which are mounted (“made available”, mapped) into containers.
* Here we can really connect to a folder, inside of the container to a folder outside of the container on your host machine.
* If you add a file on your host machine, it is accessible inside of the container, and if the container adds a file in that mapped path, it is available outside of the container in the host machine as well.
* It persists and continue to exist even if a container is shut down.
* If we add a volume to a container the volume will not be removed, even though the container is removed.

## Two Types of External Data Storages

* 1. Volumes (Managed by Docker)
  2. Bind Mounts (Managed by us)

## Volumes (Managed by Docker)

* Docker sets up a folder / path on our host machine, exact location is unknown to us.
* Managed by docker volume commands.
* The only way to access these data is via **docker volume** command
* To list all Volumes - **docker volume ls**
* If we stop our container the anonymous volume doesn’t exist anymore i.e., it exists as long our container is running.
* Hard to find the location where docker saves it.
* Usually, the Volumes are not meant to be edited by user.
* It is of two Types
  1. Anonymous Volumes
  2. Named Volumes

### Anonymous Volumes

* Anonymous Containers are removed automatically when the container runs with –rm option.
* If we start the container without –rm option, the anonymous volume would not be removed. Or even after manually removing the container.
* A new anonymous volume gets created on next run of a container and starts pilling up.

Ex – Dockerfile

FROM node

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

EXPOSE 80

// Anonymous Volume

VOLUME ["/app/feedback"]

CMD ["node", "server.js"]

* To clear the volumes, we can use

**docker volume rm <vol\_name>** or **docker volume prune**

### Named Volumes

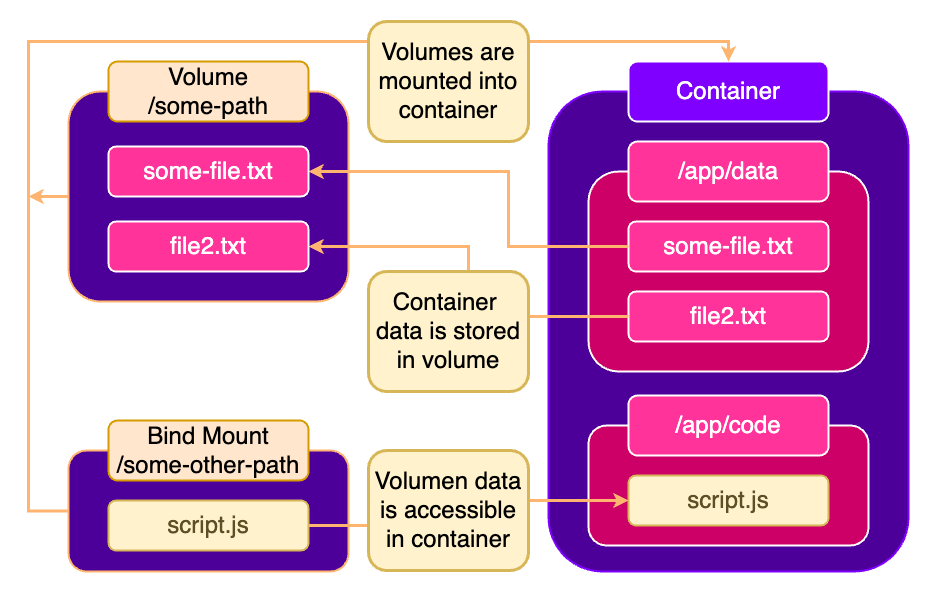
* A defined path in the container is mapped to the created volume / mount.
* Ex - /some-path on our hosting machine is mapped to /app/data
* Volumes survive even the Containers are stopped.
* Named Volumes are not attached to any container.
* To create a Named Volume - **docker run -d -p 3030:80 --rm --name feedback-app -v feedback:/app/feedback feedback-node:volume**
  1. **-v** – option stands for volume, allows to add a volume for the container
  2. **feedback:/app/feedback** – **<volume\_name>:<path\_inside\_container>** - It will create a folder inside of the container, but it will store this volume under a name chosen.

## Bind Mounts (Managed by us)

* It helps us in solving the problem to reflect the code that we changed, mostly during the development phase without building it.
* Here we set the path to which the container internal path should be mapped on our host machine.
* Since containers can read and write both to volumes, we could put our source code into such a bind mount.
* The container is aware of that the source code us not used from the copied snapshot but instead from that bind mount and thus container will have access to latest code.
* Great for persistent, editable data.
* **A Named volume can help us with persistent data but not with editable data.**
* It doesn’t affect the image, but the container.
* To Bind mount - **docker run -d -p 3030:80 --rm --name feedback-app -v feedback:/app/feedback -v "/Users/debaprasad/Documents/Project Snippets/Docker/docker-demo-4:/app" feedback-node:volumes**
* **Shortcuts to get the path - -v $(pwd):/app**
* Here we override the local folder with the container app folder.

## Understanding Container / Volume Interaction

* If we have a container and we don’t have a volume, and a bind mount, we can mount both into the container with the -V flag that means some folders inside of the container are mounted or are connected to folders on the host machine.
* If some files exist inside the container, they also exist in the outside volume and if we write a new file its also added in the folder on the host machine.
* If some files exist on volumes but not inside container, at start it loads the file from volume and that’s what we utilize with the Bind mount.



* Ex – In a Node example – (docker-demo-4) in our host machine we have the code and in the docker we run the image on the container and through the Bind mount we mount our host machine code to container app, but in our local we don’t have the node\_modules, to inorder to get that we modify the Dockerfile or modify the run command as  
  **docker run -d -p 3030:80 --rm --name feedback-app -v feedback:/app/feedback -v "/Users/debaprasad/Documents/Project Snippets/Docker/docker-demo-4:/app" -v /app/node\_modules feedback-node:volumes**

Or

FROM node

WORKDIR /app

COPY package.json .

RUN npm install

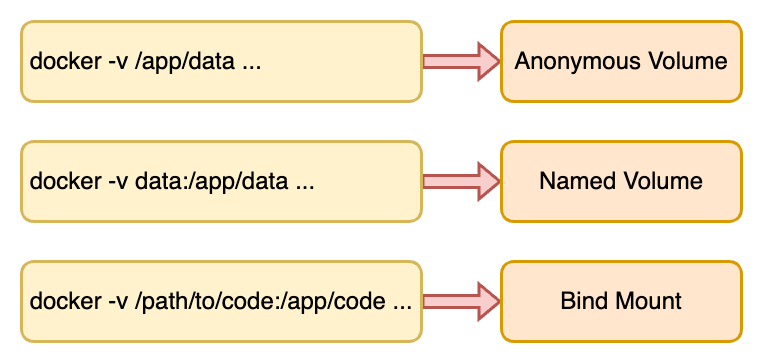
COPY . .

EXPOSE 80

# VOLUME ["/app/node\_modules"]

CMD ["node", "server.js"]

* Here what happens is that container image installs the node\_modules and since we specified the specific path node\_modules overwrites the non existant node\_modules and being used



A group of rectangular boxes with text

Description automatically generated

* Adding **:ro** makes it readable only

**docker run -d -p 3030:80 --rm --name feedback-app -v feedback:/app/feedback -v "/Users/debaprasad/Documents/Project Snippets/Docker/docker-demo-4:/app:ro" -v /app/node\_modules feedback-node:volumes**

* Now to allow specific files to be write also, and other read only

**docker run -d -p 3030:80 --rm --name feedback-app -v feedback:/app/feedback -v "/Users/debaprasad/Documents/Project Snippets/Docker/docker-demo-4:/app:ro" -v /app/temp -v /app/node\_modules feedback-node:volumes**

## Managing Docker Volumes

* Bind Mount Volumes are not manged by Docker.
* Named and Anonymous volumes are manged by Docker.
* We can create Volumes too manually else Docker creates volumes if not available during run of the container with name given or anonymous – **docker volumes create <vol\_name>**
* To inspect a volume – **docker volume inspect <vol\_name>**
* Here Docker creates the volume in Mac – "Mountpoint": "/var/lib/docker/volumes/feedback/\_data",
* However, this path is not the actual path, Docker creates this virtual machine set up, it is hard to find out where this is stored.

## COPY vs Bind Mount

* We could remove COPY command from Dockerfile, if we use Bind Mount, but in the production, we will not run with Bind Mount we might ensure that data survives through other volumes.
* Bind Mount is good for Local development.
* COPY doesn’t create any side effect for Bind Mount.

### .dockerignore file

* This tells the docker the set of files/folders not to copy.

## ARGuments and ENVironment Variables

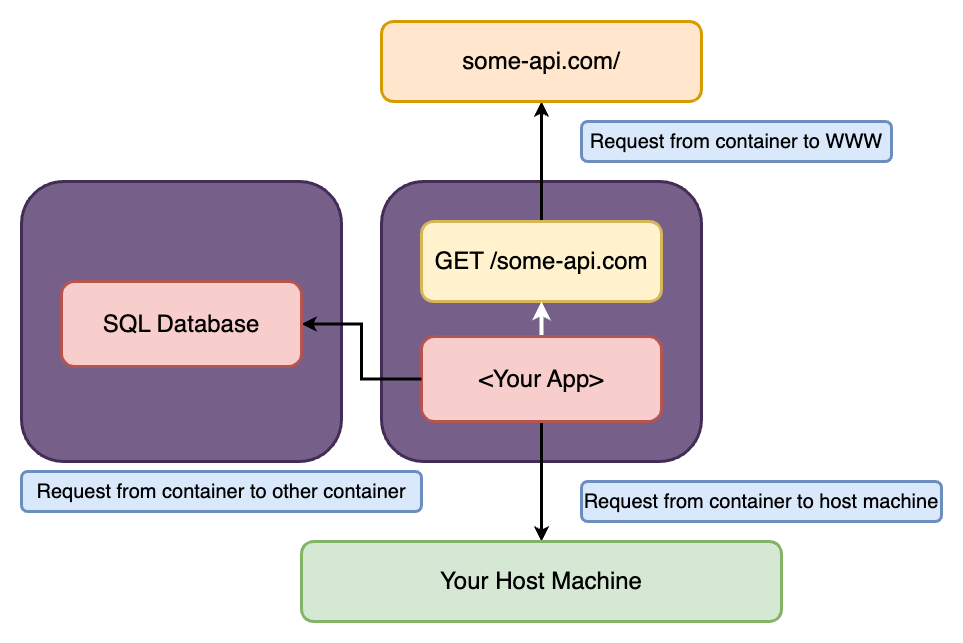
* Two options that can be set in Dockerfile, Docker build and run commands.
* Docker supports build-time ARGuments and runtime ENVironment variables.
* Allows us to create flexible images and containers because we don’t have to hardcode everything, rather set dynamically during running.
* ARG
  1. Available inside of Dockerfile, Not accessible in CMD or any application code.
  2. Set on image build via –build-arg
  3. Can be used only in Dockerfile and can’t be used in every instructions
* ENV
  1. Available inside of Dockerfile & in application code
  2. Set via ENV in Dockerfile or via –env on docker run

To pass the env variable during run - **docker run -d -p 3030:8000 --env PORT=8000 --rm --name feedback-app -v feedback:/app/feedback -v "/Users/debaprasad/Documents/Project Snippets/Docker/docker-demo-5:/app:ro" -v /app/temp -v /app/node\_modules feedback-node:env**

To pass the env file **- docker run -d -p 3030:8000 --env-file ./.env --rm --name feedback-app -v feedback:/app/feedback -v "/Users/debaprasad/Documents/Project Snippets/Docker/docker-demo-5:/app:ro" -v /app/temp -v /app/node\_modules feedback-node:env**

# Networking: (Cross-) Container Communication

* **Container to WWW:** Sending an HTTP request to some other website or Web API
  1. Don’t need any specialized configuration to work
* **Container to Host machine:** Communicating from inside your container to some other service running on your host machine.
* **Container to another container**



* When an application is running in host machine and to connect the container to host machine replace **localhost** to **host.docker.internal** and docker will do the rest.
* mongoose
* .connect("mongodb://host.docker.internal:27017/swfavorites")
* .then(() => app.listen(3000))
* .catch((err) => console.error("Error connecting to MongoDB:", err));
* To connect to other container from container, find the config by running **docker container inspect <container\_name>**, find the **NetworkSettings** and inside it locate **"IPAddress": "172.17.0.2"**, and replace the localhost to the IPAddress.
* mongoose
* .connect("mongodb://172.17.0.2:27017/swfavorites")
* .then(() => app.listen(3000))
* .catch((err) => console.error("Error connecting to MongoDB:", err));

## Docker Networks

* We can create a container network inside the Docker called networks.
* It allows communication between containers for a multiple container scenario.
* --network options helps in doing that and helps in automatically IP look up and resolving stuff.

A diagram of a network

Description automatically generated

* Unlike Volumes, Docker will not create Networks automatically.
* To create a network - **docker network create favorites-net**
* Internal Docker network that helps container to talk to each other.
* Now you can put the container name in place of localhost to connect.
* mongoose
* .connect("mongodb://mongodb:27017/swfavorites")
* .then(() => app.listen(3000))
* .catch((err) => console.error("Error connecting to MongoDB:", err));
* Docker doesn’t go inside sees the code between containers, rather solves the network connections.
* Docker Networks support different kinds of **Drivers** which influence the behavior of the Network.
* The default driver is **bridge** driver as applicable for majority of scenarios.
* The driver can be set when a network is created, simply by adding **--driver** option.

**docker network create –driver bridge my-net**

* 1. **host** – for standalone containers, isolation between container and host system is removed (i.e., they share localhost as a network)
  2. **overlay** – Multiple Docker daemons (i.e., Docker running on multiple different machines) can connect with each other. Only works in “Swarm” mode which is a dated / almost deprecated way of connecting multiple containers.
  3. **macvlan** - You can set a custom MAC address to a container – this address can then be used for communication with that container
  4. **none** - All networking is disabled.
  5. **Third-party plugins** - You can install third-party plugins which then may add all kinds of behaviors and functionalities.

# Docker Compose

* Automating multi-container setup.
* It allows you to bring up the entire setup, with all the different containers and their individual configurations with just one command.
* It is a tool that allows us to replace Docker build and Docker run commands (multiple build & run) with just one configuration file and then a set of orchestration commands to start all these containers at once and build all necessary images, & then also can use one command to stop everything.
* Docker compose will not replace Docker files for custom images rather works together.
* Docker compose does NOT replace Images or Containers; it just makes working with them easier.
* Docker compose is NOT suited for managing multiple containers on different host’s rather same host.
* To start adding Docker compose, define a docker compose file i.e., **docker-compose.yaml**

## docker-compose.yaml

* It needs to be installed in Linux separately.
* version – the version of the docker-compose specifications we wanna use and version we defined here influences the features we can use in this compose file because the docker compose specification and therefore the syntax we must use in this file is under active development and may change over time.
* We can lock so that we know which features are and are not available.
* Default the containers will be removed when container is stopped.
* While executing the docker compose, Docker will automatically create a new environment for all the services specified and it will add all the services to that network out of the box.
* Anonymous volumes and bind mounts don’t need to be specified, However, named volumes need to be specified and needs to be listed like   
  **volumes:**

**data:**

* By doing this Docker is aware of named volumes and other containers can use the same volume if they wish to
* To start all the services – **docker-compose up**
* To start all the services in detached mode – **docker-compose up -d**
* To stop all the services – **docker-compose down**
* **Note**: when docker stopped, it will delete all the networks, containers and images but not volumes.

# Commands

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl** | **Command** | **Description** | **Parameter / remember** |
| 1 | **docker build .** | To build an image |  |
| 2 | **docker run -p <system\_port>:<container\_port> <build\_key>** | To run the container | **- run** always starts a new container  **- <system\_port> :** port available to outside world to connect with container.  **- <container\_port> :** container port available where the application is running  - In the Dockerfile  EXPOSE <port\_number> is optional. |
| 3 | **docker ps** | To show list of containers | **docker ps -a –**  **-a** will show all the processes all the containers Docker created for us. |
| 4 | **docker stop <container\_name>** | To stop the container |  |
| 5 | **docker run node Ex - node** | To Run an available image |  |
| 6 | **docker run -it node** | To run the interactive terminal |  |
| 7 | **docker –help** | To see all available options |  |
| 8 | **docker start <container\_name>** | To start an existing stopped container | This doesn’t block the terminal unlike docker run |
| 9 | **docker run -p 8000:80 -d <image\_id>** | To run the container in detached mode | - **-d** represents for detached mode.  - Executing this code gives the container id, if you wanted to attach to it again. |
| 10 | **docker attach <container\_name / id>** | To attach again to a container |  |
| 11 | **docker logs <container\_name>**  **docker logs -f <container\_name>** | Fetch the logs of a container to the point of execution | **-f** - Fetches the Logs and attach to the container |
| 12 | **docker start -a -i** **<container\_name>** | To start the container in attached and interactive mode |  |
| 13 | **docker rm <container\_id>**  **or**  **docker container prune** | To delete container | post stopping the container.  **To remove unused stopped container** |
| 14 | **docker images** | To see the list of images |  |
| 15 | **docker rmi <image\_id>** | To delete the images. |  |
| 16 | **docker image prune** | To delete all the images not being used at once. |  |
| 17 | **docker run -p 3000:80 -d –rm <image\_id>** | Automatically remove the container when it exits. | --rm is the helper added |
| 18 | **docker image inspect <image\_id>** | To inspect images. |  |
| 19 | **docker cp <source\_folder\_path> <container\_name/container\_path>**  **docker cp <container\_name/container\_path> <source\_folder\_path>** | To copy file into or out of a running container | From outside to inside container  From inside of container to outside  docker cp |
| 20 | **docker build -t goals:latest .** | Name and optionally a tag in the 'name:tag' format. | -t helper is used |
| 21 | **docker run -p 3000:80 -d –rm –name <user\_provided\_container\_name> <image\_id>**  **docker run -p 3000:80 -d –rm –name <user\_provided\_container\_name> <image\_name>:<tags>** | Name a container. | --name helper is used |
| 22 | **docker tag <old\_name>:<tag> <new\_name>:<new\_tag>** | Renaming a Tag | **Tag** |
| 23 | **docker push IMAGE\_NAME** | To share the Image to any registry |  |
| 24 | **docker pull IMAGE\_NAME** | To pull the Image from the registry |  |
| 25 | **docker login** | To Establish a connection with your system and Docker hub |  |
| 26 | **docker volume ls** | To list all Volumes |  |
| 27 | **docker run -d -p 3030:80 --rm --name feedback-app -v feedback:/app/feedback feedback-node:volume** | To create a Named Volume | **-v** – option stands for volume, allows to add a volume for the container  **feedback:/app/feedback –** It will create a folder inside of the container, but it will store this volume under a name chosen |
| 28 | **docker volume rm <vol\_name>** or **docker volume prune** | To clear the volumes | Prune clears all unused volume |
| 29 | **docker run -d -p 3030:80 --rm --name feedback-app -v feedback:/app/feedback -v "/Users/debaprasad/Documents/Project Snippets/Docker/docker-demo-4:/app" feedback-node:volumes** | To Bind mount | **-v –** again to give the absolute path of the project directory, followed by the WORKDIR  Shortcuts to get the path – Mac / Linux - **-v $(pwd):/app**  Windows - -v “%cd%”:/app |
| 30 | **docker run -d -p 3030:80 --rm --name feedback-app -v feedback:/app/feedback -v "/Users/debaprasad/Documents/Project Snippets/Docker/docker-demo-4:/app" -v /app/node\_modules feedback-node:volumes** | to prioritize container-internal paths higher than external paths. Here to use the node\_modules from the volume | **-v /app/node\_modules** |
| 31 | **docker run -d -p 3030:80 --rm --name feedback-app -v feedback:/app/feedback -v "/Users/debaprasad/Documents/Project Snippets/Docker/docker-demo-4:/app:ro" -v /app/node\_modules feedback-node:volumes** | Adding **:ro** makes the Bind Mount Read only |  |
| 32 | **docker run -d -p 3030:80 --rm --name feedback-app -v feedback:/app/feedback -v "/Users/debaprasad/Documents/Project Snippets/Docker/docker-demo-4:/app:ro" -v /app/temp -v /app/node\_modules feedback-node:volumes** | To allow certain folders to be write as well by the concept of point 30 |  |
| 33 | **docker volumes create <vol\_name>** | To create volume |  |
| 34 | **docker volume inspect <vol\_name>** | To inspect volume |  |
| 35 | **docker run -d -p 3030:8000 --env PORT=8000 --rm --name feedback-app -v feedback:/app/feedback -v "/Users/debaprasad/Documents/Project Snippets/Docker/docker-demo-5:/app:ro" -v /app/temp -v /app/node\_modules feedback-node:env** | To pass the Env | **--env or -e** |
| 36 | **docker run -d -p 3030:8000 --env-file ./.env --rm --name feedback-app -v feedback:/app/feedback -v "/Users/debaprasad/Documents/Project Snippets/Docker/docker-demo-5:/app:ro" -v /app/temp -v /app/node\_modules feedback-node:env** | To pass the env as a file |  |
| 37 | **docker history <image>** | To read the values inside the image |  |
| 38 | **docker build -t feedback-node:arg --build-arg DEFAULT\_PORT=7575 .** | To pass argument while building image |  |
| 39 | **docker container inspect <container\_name>** | To inspect a container |  |
| 40 | **docker network create <net\_name>** | To create network |  |
| 41 | **docker run -d --name mongodb --network favorites-net mongo** | To run the Container with defined network |  |
| 42 | **docker network create –driver bridge my-net** | The driver can be set when a network is created, simply by adding **--driver** option. |  |

# Questions

1. **What’s a Volume (When working with Docker)?**
   1. Any Folder / file inside of a Docker container
   2. **A folder / file inside of a Docker container which is connected to some folder outside of the container.**
   3. A physical disk which is used to store Docker containers.
2. **Which Statement is correct?**
   1. Volumes are managed by Docker; you decide which folder outside of a container is mapped to a container-internal path.
   2. Volumes can only be attached to images, not to containers – except for Anonymous Volumes.
   3. **Volumes are managed by Docker; you don’t necessarily know where the host folder (which is mapped to a container-internal path) is.**
3. **What’s true about Anonymous Volumes?**
   1. **They are removed when a container, that was started with –rm is stopped.**
   2. They are never removed
   3. They are always removed when a container stops.
4. **What’s the advantage of Named Volumes?**
   1. They are easier to identify when executing docker run.
   2. You know the path on your host machine, this volume is bound to.
   3. **They survive container removal.**
5. **What’s a Bind Mount?**
   1. A path inside of the container, which is specified by you, not by Docker.
   2. **A path on your host machine, which you know and specified, that is mapped to some container-internal path.**
   3. A volume which has a name but is still fully managed by Docker.
6. **What’s typical use-case for a Bind-Mount?**
   1. You want to ensure data persists.
   2. **You want to provide live data to the container (no rebuilding needed).**
   3. You want to provide live data to the container (image rebuilding needed).
7. **Are Anonymous Volumes useless?**
   1. Yes!
   2. Yes, unless you build extremely complex apps.
   3. **No, you can use them to prioritize container-internal paths higher than external paths.**
8. **When Running containers on your system (via docker run): Can these containers communicate with the world wide web?**
   1. **Yes**
   2. No
   3. Yes, if you add **--allow-network** as an option to docker run
9. **How can applications running in a container communicate to your local host machine?**
   1. By sending requests to **localhost**
   2. By sending requests to **localhost-docker**
   3. **By using the special *host.docker.internal* address**
10. **How can containers communicate with other containers?**
    1. They can’t
    2. **By manually finding the IP addresses or by using a network**
    3. Only by exposing containers to the WWW
11. **How can containers communicate with each other if they are in the same network?**
    1. You must use the container IP address
    2. You must define address names in advance when creating the network
    3. **You can use the container names as addresses.**

# Best Practice

1. Every Container Should just do one main thing.