End to end stack example:

Web Server – NodeJS

Database – MongoDB

Messaging – Redis

Orchestration - Ansible

Containers are completely isolated environments as they can have their own processes, services, network interfaces, their own mounnts, just like virtual machine, except they all share the same OS kernel.

Different type of containers: LXC(utilized by Docker), LXD, LXCFS etc

For different Linux distros the OS Kernel it’s the same, only the customer software on top of the Kernel OS diferentiate the distros

Docker can run any flavour of OS on top of it as long as they are all based on the same Kernel (Linux). Each container has only the different software component that differentiate the distros and Docker utilizes the underlying kernel of the host (ubuntu in our case) which works with other linux distros.

Docker image is a package or a template (like a VM template) that is used to create one or more containers

Containers are running instances of images that are isolated that and have their own environments and set of processes

Install docker engine: <https://docs.docker.com/engine/install/ubuntu/>

Docker hub: hub.docker.com

Install whalesay: sudo docker run docker/whalesay cowsay Hello-World!

Basic Docker comands:

docker run <image> – used to run a container from an image (from the local host and if the image does not exist it will go to dockerhub to pull the image)

docker ps – list all running containers and some basic info about them (container ID, name of the image, current status and the name of the container)

-a (all containers)

docker stop <container\_name>

docker rm <container\_name/id> <../id> remove a stopped or exited container permanently

List the images present on our host: docker images

To remove an image: docker rmi <image> (ensure that no containers are running off of that image – stop and delete the containers before)

Only to download the image without running it: docker pull <image>

Example: docker run ubuntu

Containers are not meant to host an OS, are meant to run a specific task or process. Once the task is complete, the container exits. The container only lives as long as the process inside it is alive – if the web service for example from within the container crashes or it is stopped, the container exits.

docker run ubuntu sleep 5

To run a command on a running container: docker exec <container\_name/id> cat/etc/hosts

To run a container in the background(detach mode): docker run –d <repo>

If you would like to foreground/attach back to the running container: docker attach <container\_name/id>

The official images are stored by default in the library/ repository

If you are using your own repo: <user\_name>/repository

Enter interactively in a container: docker run –it <image> bash

-i is mapping the standard input

-t is mapping the terminal

-it interactive -> basically runs the container and connects me to that particular container

Exit code depends on the manner of exiting the container (0 if it exits automatically and other codes – 137 for example if it was stopped)

To run a contaier by an image and name the container:

docker run -d --name webapp nginx:1.14-alpine

docker run <image>:<tag> (could be a version or could be latest, by default no tag = latest)

Mapping ports: docker run –p (the desired port – host port )80:5000(the port that is already listening on – container port)

By default the container gets an internal ip, but it could be reachable through the ip of the docker host

**Volume mapping:**

Run a mysql container -> when data is generated the data files are stored in /var/lib/mysql -> as the docker container has it s own isolated file system and any changes to files happen within the container

To avoid data loss when the container is stopped, or crashes – persistent volume outside the container to the docker host you would have to map a directory outside the container on the docker host to a directory inside the container:

docker run –v /opt/datadir:/var/lib/mysql mysql

To see additional details about a specific container: docker inspect <container\_name/id> - all details of the container in a JSON format

To see logs: docker logs <name/id>

By default you are using the root user in docker engine, thus, for volume mapping you would have to:

docker run –p 8080:8080 –v /root/my-jenkins-data:/var/jenkins\_home –u root jenkins

**Docker images – layered architecture**

First you have to create an architecture requirments of the application in the right order:

1. OS – Ubuntu – layer 1

FROM Ubuntu

2. Update the source repositories using command apt repo – layer 2

RUN apt-get update

3. Install dependencies using apt – layer 2

RUN apt-get install python

4. Install Python dependencies using pip – layer 3

RUN pip install flask

RUN pip install flask-mysql

5. Copy over the source code of my application to a location like /opt folder – layer 4

COPY . /opt/source-code

6. Run the web server using „flask” command – layer 5

ENTRYPOINT FLASK\_APP=/opt/source-code/app.py flask run

The Dockerfile would look like:

**FROM Ubuntu** #start from a base OS or another image that was created before BASED on an OS

**RUN apt-get update** #instructs docker to run a particular command on the base image

**RUN apt-get install python**

**RUN pip install flask**

**RUN pip install flask-mysql**

**COPY . /opt/source-code** #copies files from the local system onto the docker image (currently on the current folder and I will copy it over to the location /opt/source-code inside the docker image)

**ENTRYPOINT FLASK\_APP=/opt/source-code/app.py flask run** #allows us to specify a command that will be runned when the image is runned as a container

Other commands for an Docker Image:

**EXPOSE 8080** #define the port that the web application is going to run within the container

**WORKDIR /opt**

**CMD <command> <param1> (CMD sleep 5)**

**CMD [<„command”>,<|”param1”]> (CMD [„sleep”, „5]**

**ENTRYPOINT [„sleep”] – The entrypoint instruction is like the command instruction as you can specify the program that will be run when the container starts and whatever you specify on the command line will get appended in the entrypoint, for example:**

**Normally, you would: docker run <image> sleep 10 – but if you have sleep in your entrypoint:**

**docker run <image> 10**

**If you don t specify a time for the sleep command it will not work as it has not a default value, thus cmd and entrypoint can be used together:**

**ENTRYPOINT [„sleep”]**

**CMD [„5”]**

**The default value introduced by cmd can be overwritten by the commandline: docker run <image> 10**

**To override the entrypoint command: docker run –entrypoint sleep2.0 <image> 10**

To build the image: docker build Dockerfile –t mmumshad/my-custom-app(name of the image)

-t = tag and allows you to give a name

To make it available on the public hub registry: docker push mmumshad/my-custom-app (but the image when build should have been tagged with the user name - you have to be logged in)

Log in to docker: docker login

mmumshad(name of the account)/my-custom-app(name of the image)

Docker file is a text file written in a specific format that Docker can understand – it’s in an instruction (everything from the left in caps is an instruction: FROM, RUN, COPY, ENTRYPOINT) and argument format (everything on the right are arguments to those instructions)

To see which of the 5 layers consume more or details, you may run:

docker history <image-name>

When you run the docker build command, you could see the various steps involved and the result of each task. All the layers build are cast so the layered architecture helps to restart docker build from that particular step in case it fails or if you were to add new steps in the build process you would not have to start all over again.

All the layers build are cashed by docker and if the Layer 3 fails and you were to fix the issue and rerun docker build it will reuse the previous layers from cache and continue to build the remaining layers (it works even when adding other layers – it is faster).

Run an instance of the image webapp-color and publish port 8080 on the container to 8282 on the host.

docker run -p 8282:8080 webapp-color

docker run –p <host\_port to be exposed>:<container port which is going to be mapped> <name>

To pass a environment variable within the container: docker run –e APP\_COLOR=blue <image\_name>

Inspect environment variable on a container that is already running: docker inspect <container\_name>

which will print under the config section the list of environment variables set for the container

docker inspect b03 | grep -A 2 Env # -A argument prints a number of lines

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:  docker exec -it blue-app env

**Docker Compose**

If we need to setup a complex app running multiple services a better way to do it is with Docker Compose – we could create a config file in yaml format named docker-compose.yml

docker-compose.yml:

services:

web:

image: „<image>”

database:

image: „mongodb”

messaging:

image: „redis:alpine”

orchestration:

image: „ansible”

To bring the whole app stack: **docker-compose up**

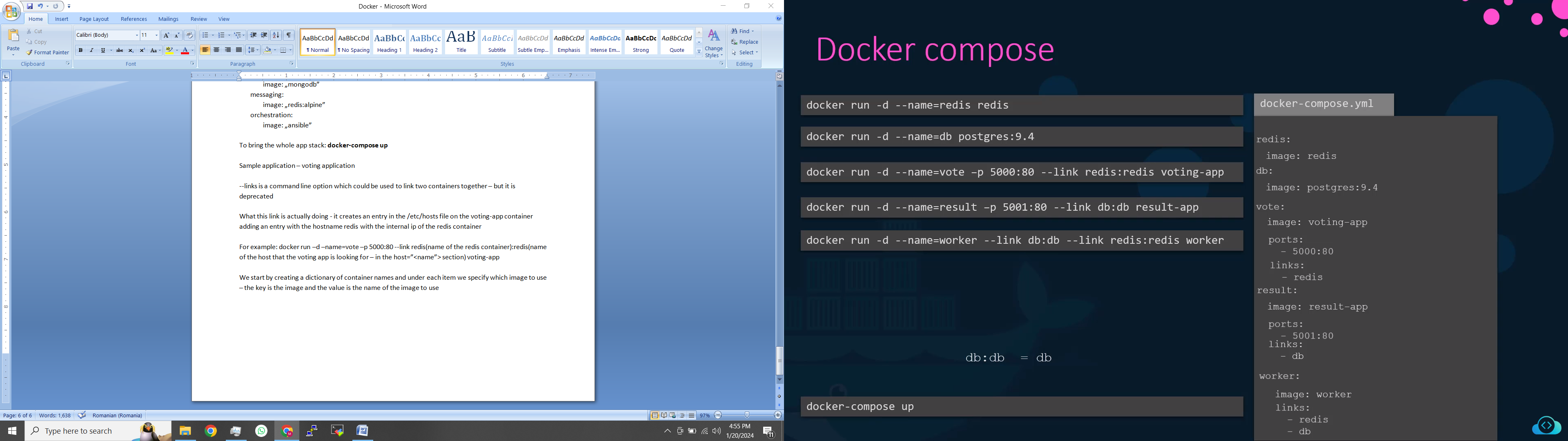
Sample application – voting application

--links is a command line option which could be used to link two containers together – but it is deprecated

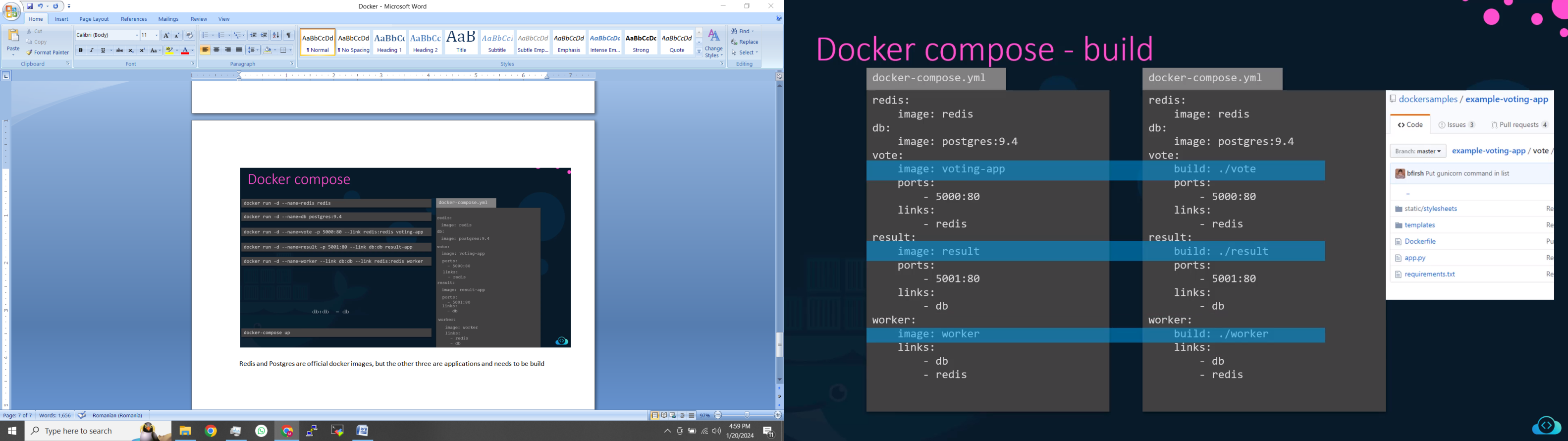
What this link is actually doing - it creates an entry in the /etc/hosts file on the voting-app container adding an entry with the hostname redis with the internal ip of the redis container

For example: docker run –d –name=vote –p 5000:80 --link redis(name of the redis container):redis(name of the host that the voting app is looking for – in the host=”<name”> section) voting-app

We start by creating a dictionary of container names and under each item we specify which image to use – the key is the image and the value is the name of the image to use



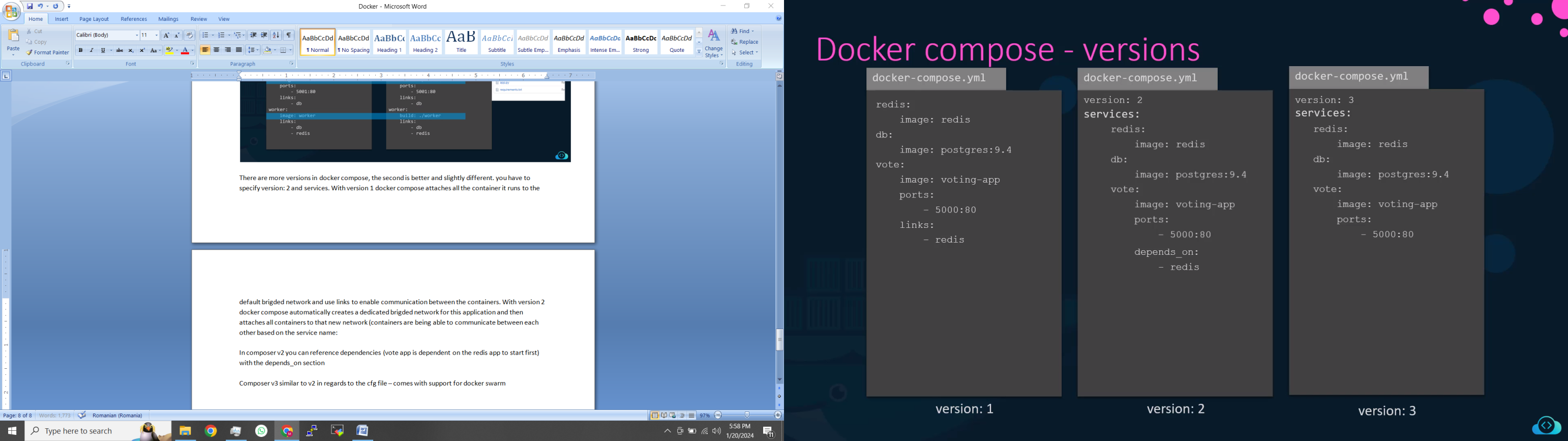
Redis and Postgres are official docker images, but the other three are applications and needs to be build



There are more versions in docker compose, the second is better and slightly different. you have to specify version: 2 and services. With version 1 docker compose attaches all the container it runs to the default brigded network and use links to enable communication between the containers. With version 2 docker compose automatically creates a dedicated brigded network for this application and then attaches all containers to that new network (containers are being able to communicate between each other based on the service name:

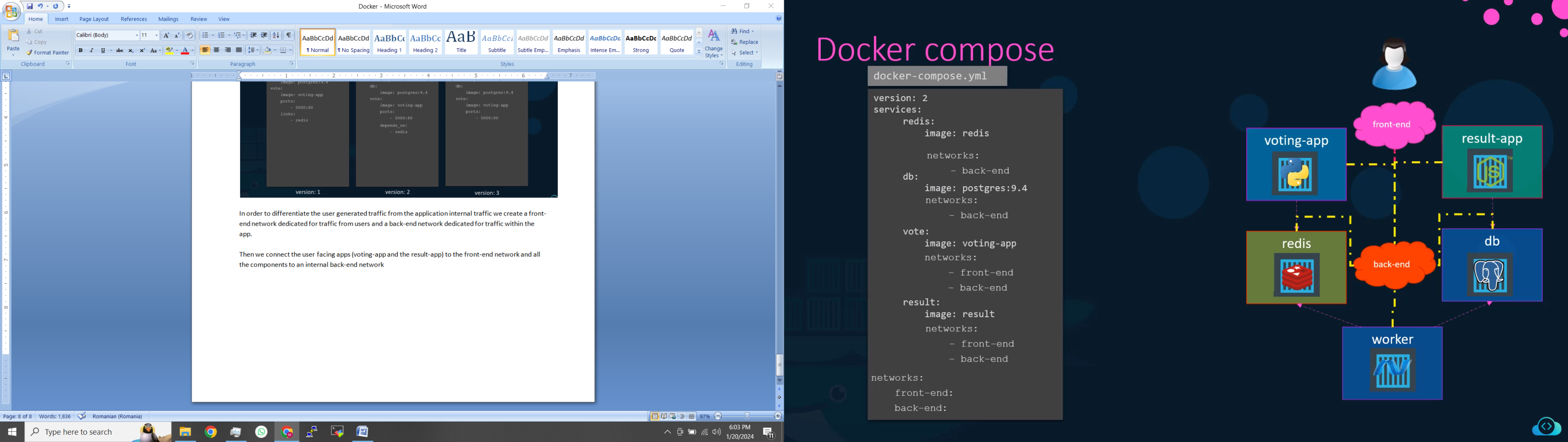
In composer v2 you can reference dependencies (vote app is dependent on the redis app to start first) with the depends\_on section

Composer v3 similar to v2 in regards to the cfg file – comes with support for docker swarm



In order to differentiate the user generated traffic from the application internal traffic we create a front-end network dedicated for traffic from users and a back-end network dedicated for traffic within the app.

Then we connect the user facing apps (voting-app and the result-app) to the front-end network and all the components to an internal back-end network



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

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

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

Docker engine – a host with docker installed on it:

Docker CLI – command line interface – it uses the REST API to interact with the Docker daemon – it may not be necessary on the same host and can be used from another system and can still work with a remote docker engine

docker –H=remote-docker-engine:2375

docker –H=10.123.2.1:2375 run nginx

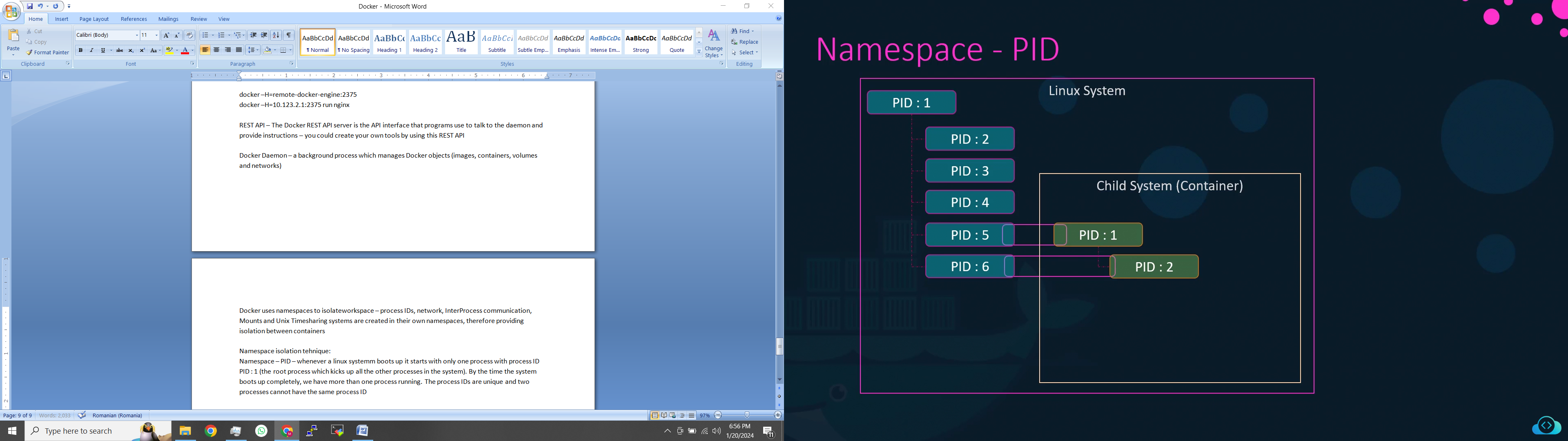
REST API – The Docker REST API server is the API interface that programs use to talk to the daemon and provide instructions – you could create your own tools by using this REST API

Docker Daemon – a background process which manages Docker objects (images, containers, volumes and networks)

Docker uses namespaces to isolateworkspace – process IDs, network, InterProcess communication, Mounts and Unix Timesharing systems are created in their own namespaces, therefore providing isolation between containers

Namespace isolation tehnique:

Namespace – PID – whenever a linux systemm boots up it starts with only one process with process ID PID : 1 (the root process which kicks up all the other processes in the system). By the time the system boots up completely, we have more than one process running. The process IDs are unique and two processes cannot have the same process ID



It s a trick – because those processes are running in the host as well, but they are projected through namespaces with other project ID in the container

The underlying Docker Host as well as the containers share the same system resources (cpu, mem). By default there is no restriction in regards to resource consumption made by a container, thus, a container may exhaust the resources on the underlying host.

You can restrict the amount of mem/cpu a container can use. Docker uses cgroups (control groups) to restrict the amount of hardware resources allocated to each container and can be done:

docker run --cpus=.5 <image> (the container will not use more than 50% of the host CPU at any given time)

docker run --memory=100m <image>

References:

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

<https://docs.docker.com/engine/api/>

<https://docs.docker.com/config/containers/runmetrics/#control-groups>

Docker storage:

Docker stores data on the local file system: when you install docker on a system it creates a folder structure in /var/lib/docker where docker stores by default data in the aufs, containers, image and volumes folders.

This is done through the layered architecture principle – if you have the same layers in different Dockerfiles, let’s say the only difference is in the source code, Docker is not going to build the first three layers, instead it’s going to reuse the same layers it built for the first application from the cache and buils only the new lines (faster, efficient and saves space).

Let’s say we have a docker image with 5 image layers – they are read only. When you run the container, another layer is created (container layer) which is writable (read & write) in order to store data from the container such as log files used by the applications, temporary files generated by the container or just any file modified by the user on that container. The life of this layer is only during the container is alive – when the container is destroyed the layer and all the changes stored in it are destroyed

To persist data (working with databases requires data preservation): add a persistent volume to the container: **docker volume create <volume\_name>**

Volume mounting: mount the volume in a docker container (this command automatically creates a volume):

**docker run –v <volume\_name>:/var/lib/mysql(the location inside my container which is the default location where mysql stores data) <image>**

Bind mounting: If I had my external storage on the Docker host at /data an I would like to store in there the data and not in the default /var/lib/volumes folder – we have to provide the complete path to the folder we would like to mount (/data/mysql):

**docker run-v /data/mysql:/var/lib/mysql <image>**

The preferred way instead –v is --mount:

docker run \

--mount type=bind,source=/data/mysql(location on my host),target=/var/lib/mysql(location on my container) <image>

Storage drivers enablelayered architecture. Maintaining the layered architecture, creating a writable layer, moving files across layers to enable copy&write etc it’s performed by the storage drivers.

Storage drivers: (AUFS, ZFS, BTRFS, Device Mapper, Overlay, Overlay2) – depends which one is chosen based on the OS – Ubuntu: AUFS

References:

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

<https://docs.docker.com/storage/storagedriver/>

<https://docs.docker.com/storage/volumes/>

When you install Docker, it creates three default networks:

**Bridge** – the default network a container gets attached to. It’s a private internal network created by Docker on the host – containers get by default internal IP address, usually in the range 172.17.0.0

To access these containers you have to map their ports to ports on the Docker host.

You can create other networks other than the default bridge (172.17...):

docker network create \

--driver bride\

--subnet 182.18.0.06/16

custom –isolated-network (name)

To list all networks: docker network ls

**None**

With the none network the containers are not attached to any network, thus, the container does not have access to external network or internal (other containers) – they run in an isolated network

**Host**

Another way to access the containers externally is to associate the container to the host network – this takes out any network isolation between the docker host and the container.

To attach the container to another network: docker run <image> --network=<network\_name>

To see more information in regards to networking of a container: docker inspect <container\_id/name>

To see more information in regards to a network: docker network inspect <network\_name>

Containers can reach themselves using their names(built in DNS server which runs at 127.0.0.11)

Docker isolates the containers within the host by using network namespaces that create a separate namespace for each container, it then uses virtual ethernet pairs to connect containers together.

Create a new network named wp-mysql-network using the bridge driver. Allocate subnet 182.18.0.1/24. Configure Gateway 182.18.0.1 : **docker network create --driver bridge --subnet 182.18.0.1/24 --gateway 182.18.0.1 wp-mysql-network**

Deploy a web application named webapp using the kodekloud/simple-webapp-mysql image. Expose the port to 38080 on the host.  
  
The application makes use of two environment variable:  
1: DB\_Host with the value mysql-db.  
2: DB\_Password with the value db\_pass123.  
Make sure to attach it to the newly created network called wp-mysql-network.

Also make sure to link the MySQL and the webapp container.

docker run --network=wp-mysql-network -e DB\_Host=mysql-db -e DB\_Password=db\_pass123 -p 38080:8080 --name webapp --link mysql-db:mysql-db -d kodekloud/simple-webapp-mysql

Docker registry – central repository of docker images

image: docker.io(registry)/ nginx(user/account)/nginx( image/repository name)

Popular registry:

Docker hub: docker.io

Google registry: gcr.io (a lot of k8s images are stored)

Log into your private registry: docker login private-registry.io

Run the image after logged in: docker run private-registry.io/apps/internal-app

Deploy your own private registry within your organization (docker registry as an image):

**docker run –d –p 5000:5000 –name registry registry:2**

To push your own image to your own – you have to tag the image with the private registry url in it:

**docker image tag my-image localhost:5000/my-image**

docker push localhost:5000/my-image

docker pull localhost/<IP>:5000/my-image

You can set up the restart policy for a container with: --restart=always

To check the list of images pushed , use curl -X GET localhost:5000/v2/\_catalog

Remove all images from local: docker image prune –a

**Container orchestration** is a solution that consists of a set of tools and scripts that can help host containers in a production environment: docker swarm, kubernetes, mesos(from apache)

**Docker swarm:**

To set up a docker swarm you must have host/hosts with docker installed on them and designate the host to be the Swarm Manager and other ar slaves/workers:

Swarm Manager: docker swarm init

Node worker: docker swarm join --token <token> (provided when you init the manager)

Docker services are one or more instances of a single application/service that runs across the nodes in the swarm cluster

docker service create --replicas=100 nodejs