|  |  |  |
| --- | --- | --- |
| **#** | **Command** | **Description** |
| 1 | docker run name\_of\_image | To run a docker instance |
| 2 | docker ps | List all running containers |
| 3 | docker ps -a | List all running and stop containers |
| 4 | docker stop name\_or\_id\_of\_container | Stop running container |
| 5 | docker rm name\_or\_id\_of\_container | To remove stopped container permanently |
| 6 | docker images | List downloaded images and their sizes |
| 7 | docker rmi name\_of\_image | To remove image (make sure to stop all running containers of this image before running this command) |
| 8 | docker pull name\_of\_image | Pull image and store it in our host |
| 9 | docker run name\_of\_image sleep 5 | To run commands inside a container, here the command is sleep 5. It will run the container and instead of immediately stopping it, it will run the command sleep 5 and will stop after 5 second |
| 10 | docker exec name\_of\_container command | Execute command on running container |
| 11 | docker run -d name\_of\_image | It will run the container in detach mode or in foreground. |
| 12 | Docker attach container\_id | It will attach us to running container so that we can see the output of running service. |

**Basic Commands:**

**Advanced Commands:**

|  |  |  |
| --- | --- | --- |
| **#** | **Command** | **Description** |
| **1** | docker run image\_name:version | Now it will download and run the specific version of that image. It’s called tag.  By default, docker uses the tag named latest. |
| **2** | docker run -it image\_name | **-i:** For interactive mode  **-t**: For attach with container terminal |
| **3** | docker run -p 80:8080 image\_name | **-p:** To map container port with host port  Now when we go to localhost:80 it will redirect us to localhost:8080. This way we can run multiple instances of our application. |
| **4** | docker run -v /opt/datadir:/var/lib/mysql mysql | **Volume Mapping**  To map internal directory of container to docker host directory to persist data even after the container is stopped or deleted. |
| **5** | docker inspect container\_id\_or\_name | To see advanced details about container is json format. |
| **6** | docker logs container\_name\_or\_id | To see the logs of containers running in detached mode. |

**How to create my own image?**

* We need to create **DockerFile** using provided instructions from developers about ***how to setup up application***.
* Then build your image using command:

**docker build DockerFile -t account\_name/image\_name**

* The above command will create the image locally. To make it available publicly on docker hub use the following command:

**docker push account\_name/image\_name**

**CMD command in Docker File:**

CMD command param1 e.g CMD sleep 5

CMD [“command”,”param1”] e.g CMD [“sleep”,”5”]

Whenever we run the container it will automatically execute the command written after CMD.

**Entry Point command in Docker File:**

ENTRYPOINT [“command”] e.g ENTRYPOINT[“sleep”]

Now when we execute docker run image 10, it will automatically execute sleep 10 command.

If we want to specify default value for ENTRYPOINT to be used if no argument is given from command line then:

ENTRYPOINT [“command”] e.g ENTRYPOINT[“sleep”]

CMD[“param1”] e.g CMD[“5”]

So, it will automatically execute following command when we run our container:

sleep 5

If we want to override entry point at run time then:

docker run –entrypoint sleep2.0 image param1

**Networks:**

1. Bridge(default)
2. None
3. Host

**Docker Layers:**

Docker creates a layer against every statement written in Docker file. So, if we have 5 statements then we will have 5 layers in Docker image. These layers help docker to build images efficiently. If two Docker files have same statements, then Docker will reuse the cached layers when we try to rebuild file. Also if we made any changes in code then Docker will only modified the layer containing the code and will use the cached layers while rebuilding.

When we run a container, docker will append another layer to image named as Container Layer. So, when we will try to change code during execution of container, then docker will only made those changes in Container Layer and when we destroy that container then these changes also vanished. So in order to reflect these changes permanently we can use Volumes.

docker run -v source:target image

source: Folder path where we want to store container data

target: Folder path where the docker previously store data

**Docker Compose:**

If we needed to setup a complex application using multiple services, a better way to do it is to use docker compose. With docker compose we can create a configuration file in yaml format and put them different services and options related to them. Then we can run docker-compose up command to bring up the entire application stack. It is easy to implement, run and maintain as all changes are in docker-compose file.

Instead of using docker compose we can also run these services in separate container using docker run and then can connect these using –link option as:

docker run --name worker --link source:host\_name image\_name

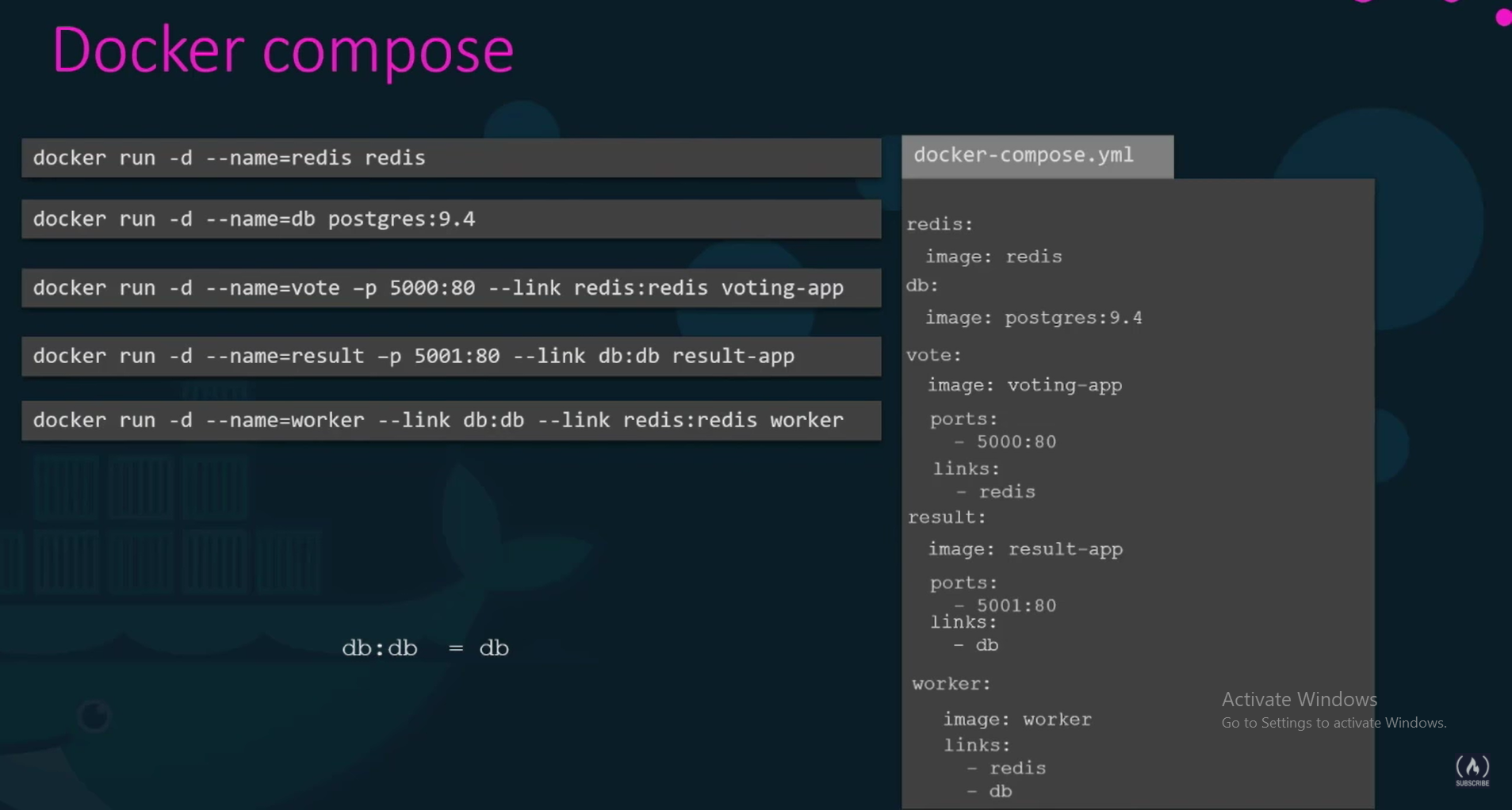
source: name of container running the required service

host\_name: The name used by this container to access this service

e.g: docker run --name worker --link redis:redis worker\_image

--link creates an entry in /etc/hosts file of worker container, with host name as redis with internal IP of redis container.

**Generate docker-compose file instead of docker run commands:**



Here we have used the image:image\_name attribute. So, when we execute docker-compose up command it will try to pull these images. If for our custom apps (like voting-app), the image is not available then we can simply use build option instead of image to instruct docker to build image instead of pulling this from docker hub.

build: path to Dockerfile containing instructions to build image

**Docker Compose Versions:**

**Version 1:**

There is no option to run containers on custom network instead of default bridge network. Also, we have dependency order. Means we need to run independent containers prior to dependent ones.

**Version 2:**

We need to mention all our application stack under services section.

For version 2 and up, we need to specify the version at top of file, so that the docker can understand the format of file.

In version 2, docker create a dedicated bridge network for all containers inside it and attach all containers to it. So, each container can communicate with each other using its service name, so no need to use links in version 2.

Also, in version 2, we can specify **depends\_on** property to mention that this service must be start after depends on service is started. So, no need to mention everything in order.

**Version 3:**

Like version 2 in format, but it also has support for docker-swarn.



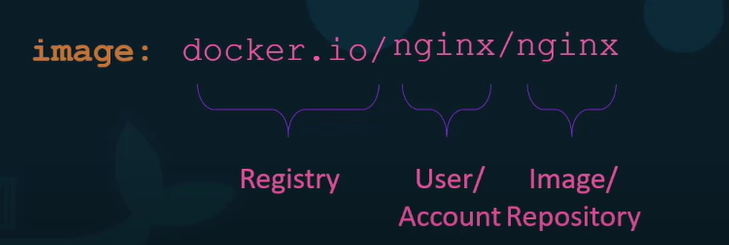
**Docker Registry:**

Where all docker images are stored.

When we run:

docker run nginx

Here the nginx name is taken as:



Default user/account name is assumed same as image name and default registry is assumed as docker hub whose DNS is docker.io

If we do not want images to be publicly available, then we can use private registry. To pull or push images to private registry we need to login using command and need to provide credentials:

docker login private-registry.io

**C Groups (Control Groups):**

By default, a container can use all of the available memory and CPU of host machine. To restrict container to use specific amount of CPU and memory, we use cgroups.

docker run --cpus=.5 ubuntu

docker run --memory=100m ubuntu

**Container Orchestration:**

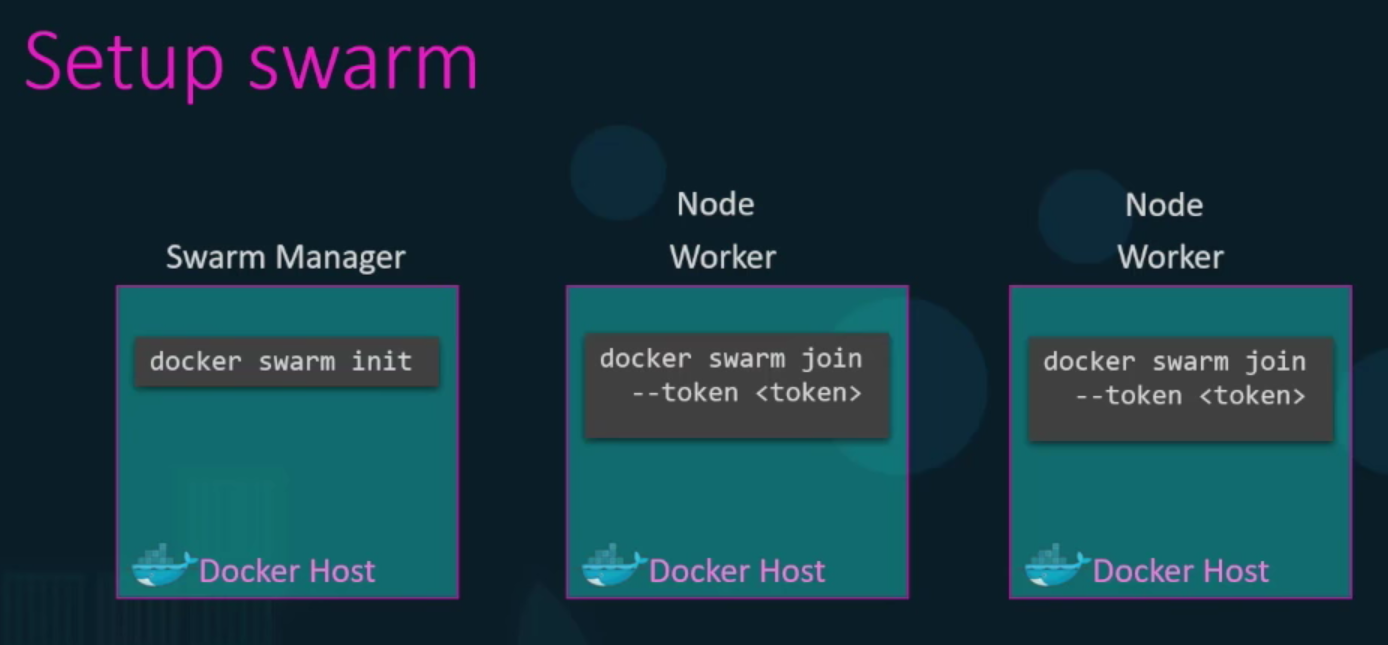
What if there are thousands of users of application, then we need to run multiple instances of application using docker run command multiple times. Also, we need to stop some instances and rerun them as traffic increases or decreases. Also, we need to take care of health of a instance, if an instance is down, then we may need to rerun this again.

This is where container orchestration comes into place. It consists of tools that help host containers in production environment. Typically, it consists of multiple docker host that can host containers so that if one fails the application is still accessible through others. It also allows you to run hundreds or thousands of instances of your application through single command. It also allows you to manage load balancing between hosts and provide option to increase or decrease host as per need. We have following common solutions available:

1. Docker Swarm
2. Kubernetes
3. Mesos

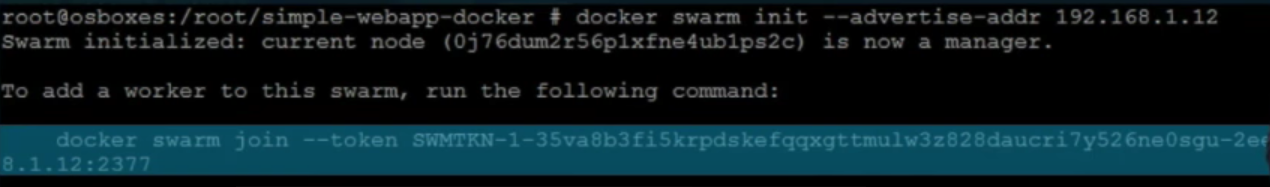
**Docker Swarm:**

We have a single swarm manager and multiple worker nodes as:

****

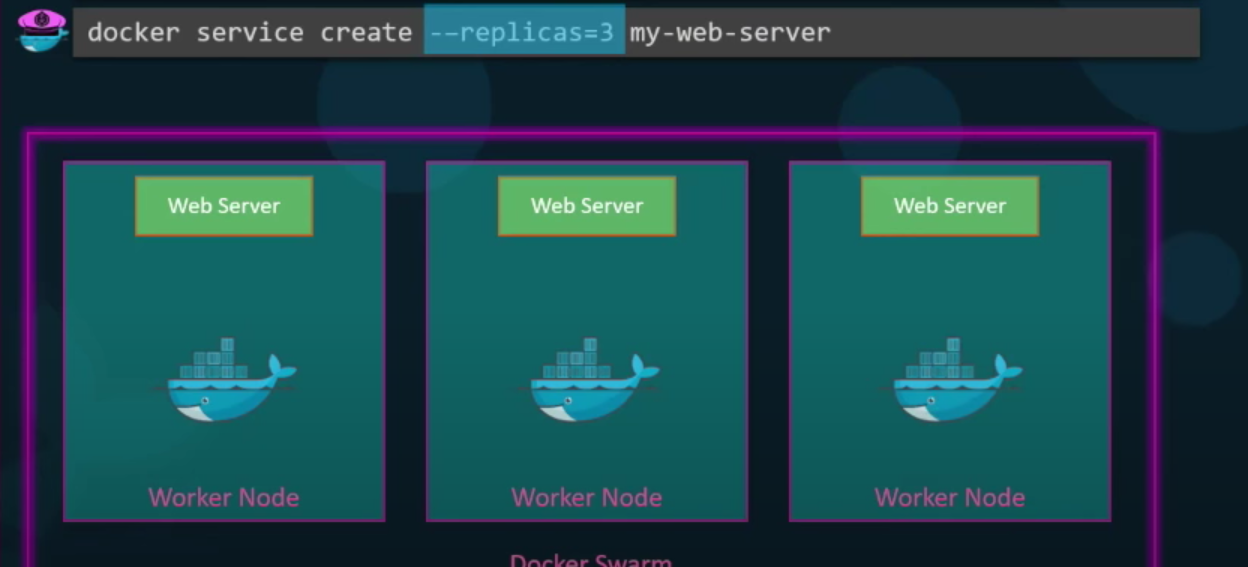
On swarm manager we execute command:

**docker swarm init**



Copy and paste the provided command on worker nodes to connect them with manager.

To run containers, execute following command on manager:



replicas=3 means that we want to execute 3 instances of my-web-server image on workers.