INCEPTION

I. Introduction - What is docker ? :

From a practical standpoint, Docker is just a way to package software so it can run on any hardware. In order to understand how this process works there are 3 main things to understand :

- A dockerfile is a blueprint / plan / model for building a docker image ;
- A docker image is a template for running docker container ;
- A docker container is simply a running process (each has a unique ID and is linked to an image).

The whole point of Docker is to reproduce environments. The developer who creates the software can define the environment with a docker file. Then, any developer, at that point, can use the docker file to rebuild the environment which is saved as an immutable snapshot known as a docker image. These images can then be uploaded to the cloud in both public and private registries. By now, any developer or server that wants to run that software can pull the image down to create a docker container that is just a running process of that image. In other words, one image file can be used to spawn the same process multiple times in multiple places. It is only at that point where tools like kubernetes and swarm come into play to scale containers to an infinite workload.

II. Docker Containers & Virtual Machines:

1. Operating Systems - OS Kernel and Softwares :

Operating systems such as Ubuntu, Fedora, etc. all consist of 2 things : an OS Kernel and a set of software.

An OS Kernel is responsible for interacting with the underlying hardware. While the OS Kernel remains the same in the operating systems listed before hand, this is to say Linux, it is the software above it that makes these operating systems different. These softwares may consist of different user interfaces, drivers, compilers, file managers, developer tools, etc.

Therefore, there is a common OS Kernel shared across all races and some custom softwares that differentiates operating systems from one another.

In the same way, containers share the same OS Kernel and have completely isolated environments. Indeed, Docker can run any flavor of OS on top of an operating system as long as they are all based on the same OS Kernel.

For example, if the underlying OS is Ubuntu, Docker can run a container based on another distribution such as Debian, Federa, etc. Therefore, a Docker container only has the additional software that makes operating systems different and Docker will utilize the underlying OS Kernel of the docker host.

By definition, a Docker container with a different OS Kernel than the docker host will not be able to run. However, there is a way of making this work by running a container on a virtual machine of the wanted OS Kernel. How does this work ?

2. Containers :

When it comes to containers, the architecture starts with an underlying hardware infrastructure, then an operating system where Docker is installed. Docker then manages the containers that run with libraries and dependencies alone.

Container		Container	
Application		Application	
Libs	Deps	Libs	Deps
Docker			
Operating System			
Hardware Infrastructure			

3. Virtual Machines :

On the other hand, when it comes to virtual machines, the structure starts a hypervisor on the hardware infrastructure and then the virtual machines on them. Virtual machines have their own operating system inside them, then they have dependencies and, then again, applications.

Virtual Machine		Virtual Machine	
Application		Application	
Libs	Deps	Libs	Deps
Operating System		Operating System	
*Hypervisor			
Hardware Infrastructure			

*A hypervisor is a computer software, firmware or hardware that creates and runs virtual machines.

4. Containers VS Virtual Machines :

	Containers	Virtual Machines
Utilization	Lightweight	Highweight (utilization of underlying resources as there are multiple virtual operating systems and OS Kernel running)
Size	Megabytes	Gigabytes
Boot Up	Fast - in matters of seconds	Slower - in a matter of minutes (as it needs to put up the entire operating system)
Isolation	Less (more resources are shared between the containers)	Complete (as virtual machines don't rely on the underlying OS or Kernel)

5. Containers & Virtual Machines :

In the case where there are large environments with thousands of application containers running thousands of Docker hosts, there is the possibility to have containers provisioned on virtual Docker hosts. That way, we can use the advantages of both technologies :

- On one hand, the benefits of virtualization in order to easily provision or decommission Docker hosts as required with virtual machines ;
- And on the other hand make use of the benefits of Docker to easily provision applications and quickly scale them as required.

Virtual Machine				Virtual Machine			
Containers		Containers		Containers		Containers	
Application		Application		Application		Application	
Libs	Deps	Libs	Deps	Libs	Deps	Libs	Deps
Docker				Docker			
Operating System				Operating System			
Hypervisor							
Hardware Infrastructure							

III. Docker Commands :

1. Basic Commands :

Commands	Explanation	Note
docker run <container name/ID> / sleep <time> /-d	Run a container from an image. Sleep will exit the command after the specified time has passed and -d will run the container in the background.	If the image is not present on the host, it will automatically go out to Docker Hub and pull the image down. However, this is only done the first time. For subsequent executions the same image will be reused.
docker ps / -a	List all running containers and some basic information about them. -a lists of all running containers as well as previously stopped or existing containers.	The information listed include the container ID, the name of the image used to run the containers, the current status and the name of the container.
docker stop <container name/ID>	Stop the specified running container.	
docker rm <container name/ID>	Gets rid of the specified container.	
docker images	Lists all available images and their sizes.	
docker rmi <image name/ID>	Removes the specified images.	This command requires to verify, beforehand, that no containers are running off of the specified image and delete all of them in order to delete the image.
docker pull <image name/ID>	Downloads the specified image and stores it on the host.	

docker exec <container name/ID> <command>	Executes the specified command on the docker container.	
docker attach <container name/ID>	Stops the containers if it is running in the background (docker run <> -d)	

2. Docker Run Commands :

Name	Command	Explanation
Tag	docker run <container name/ID>:version	Runs the specified version of the container (docker actually pulls the image of that version and runs it). If the tag is not specified, then Docker will use the latest version.
-i	docker run -i <container name/ID>	Will set the container to interactive mode (for containers that require an input in a terminal, but without a prompt)
-t	docker run -t <container name/ID>	Will attach the container to a terminal (for containers that require an input in a terminal with a prompt). This option is usually used with the interactive mode option in order to get a prompt, an output and an input).
-p	docker run -p <local host map port>: <port on docker container> <container name/ID>	Maps a local host map port to a port on the Docker container so that any user can access the application by going to the corresponding URL.
-v	docker run -v <path to a local directory>:	Runs the containers and implicitly mounts the external directory to a folder inside the

	<path of the file or directory to mounted in the container> <container name/ID>	specified container. The information of the docker will be stored locally and will remain even if the container is deleted.
inspect	docker inspect <container name/ID>	Returns all the details of the specified container such as the state mounds, configuration data, network setting, etc.
logs	docker logs <container name/ID>	Returns the logs of the container ran in the background.

IV. Docker Images :

1. Introduction - What is an image ? What is it used for ? :

An image is a package or a template just like a virtual machine template and is used to create one or more containers. Containers, on the other hand, are running instances of images that are isolated and have their own environments and set of processors.

Images are used in the containerization process, this is to say the packaging of software code. This includes an operating system (OS), all libraries and dependencies required to run the code in order to create a single lightweight executable, called a container, that runs constantly on any infrastructure.

2. Docker image naming convention :

Each image name follows the Docker image naming convention. For example, for an image <nginx>, its name is actually <nginx/nginx>. The first part that stands for the user or account name.i If an account name or repository name isn't provided it assumes that it is the same as the given name. A username is usually your Docker Hub account name or, if it is an organization, the name of the organization.

3. Image Creation :

There are lots of pre-existing containerized versions of applications readily available. Therefore, most organizations have their own products containerized and available in a public Docker or repository, more commonly known as Docker Hub or Docker Store.

The steps to create an Docker image are as follows :
- Create a Dockerfile named as precend ;
- Write down the instructions required to set up the application, such as installing dependencies, where to copy the source code from and to, what the entry point to the application is, etc. ;
- Build the Docker image with the <docker build> command and specify the Dockerfile as input as well as a tag named for the image which will create an image locally on the system ;
- Make the image available on the public Docker Hub registry, run the <docker push> command and specify the name of the image just created ;

4. <u>Dockerfile :</u>

   a. <u>Format :</u>

      A Dockerfile is a text file written in a specific format that Docker can understand, this format is in an instructions and arguments format. Instructions are uppercased texts that inform Docker to perform a specific action while creating an image. Arguments, one the other hand, are specified under a lowercase text format.

Instruction	Argument(s)	Explanation
FROM	Operating system	First instruction in any Dockerfile. Defines what the base operating system should be for the container. That operating system can either be an operating system as we speak or another image that what created prior to this one that is itself based on an operating system and not an image.
RUN	command	Instructs Docker to run a particular command on those base images. In other words, it installs all dependencies required for the Docker image.
COPY	from path & to path	Copies files from the local system onto the docker image.
ENTRY POINT	“command”	Specifies a command that runs when the image will be run itself as a container.
CMD	command param / [“command” , “param”]	Defines the program that will be run within the container when it starts

b. CMD vs ENTRYPOINT :

The <ENTRYPOINT> instruction is like the <CMD> instruction as they both specify the program that will be run when the container starts.

However, with the <ENTRYPOINT> instruction you can specify on the command line the wanted parameter: that parameter will automatically append itself to the command specified in the <ENTRYPOINT> instruction is the Dockerfile. Whereas with the <CMD> instruction, the command line parameters passed will get replaced entirely.

If there are no specified parameters in the command line when the <ENTRYPOINT> command is used and that the command requires a parameter, the corresponding error will be returned. To set a default parameter to the <ENTRYPOINT> instruction use the <CMD> instruction as well and specify only the parameter(s) of the command specified with the <ENTRYPOINT> instruction. The <CMD> instruction will be appended to the <ENTRYPOINT> instruction.

c. Architecture :

When Docker builds an image it actually builds in a layered architecture. Each line of instructions creates a new layer in the Docker image with just the changes from the previous layer. Since each layer only stores the changes from the previous layer it is reflected in the size as well. The changes in size can be seen by running the <docker history> command followed by the name or ID of the image.

While running the <docker build> command, the steps involved and the result of each task will be cast. The layered architecture helps restart Docker from a particular step in case it fails or if new steps were added in the build process : no need to start over again. In case of failure, the <docker build> command will automatically start building the image from the layer where the failure occurred (the layers that have not failed prior to this one have already been built successfully). In case of added layers, <docker build> will start rebuilding the image from the layer where the first additional instruction is located.

5. <u>Image Types :</u>

Products that can be containerized, turned into an image and build as a container :

- Databases - mysql, mango, etc. ;
- Development tools - php, gradel, etc. ;
- Operating systems - Ubuntu, WordPress, etc;
- Browsers - Google, FireFox, etc;
- Command line tool and libraries - curl, etc ;
- Music and podcast application - Spotify, Deezer, etc. ;
- Really anything containerized.

V. Docker Networking :

1. General Networks :

When downloading Docker it creates three networks automatically :

a. Bridge Network : <docker run> <image name/ID>

Bridge is the default network a container gets attached to. It is a private and internal network created by Docker on the host. All containers attached to the network by default get an internal IP address usually in the range 172.17.0 series. With this IP address, containers can access each other if required. To access any of these containers outside the world it is required to map the ports of these containers to ports on the Docker host. Another way to access those containers externally is to associate to the host network.

b. Host Network : <docker run> <image name/ID> <--network=host>

.

If you use the host network mode for a container, that container's network stack is not isolated from the Docker host (the container shares the host's networking namespace), and the container does not get its own IP-address allocated. With that said, it is not possible to run multiple web containers on the same host on the same port as the ports are now common to all containers in the host network.

c. None Network : <docker run> <image name/ID> <--network=none>

When it comes to the none network, containers are not attached to any network and don't have any access to the external network or other containers : they run in an isolated network.

2. User-defined Networks :

It is also possible to create an internal network by using the docker command <docker network create> and specify :

- The driver with the parameter <–driver> ;
- The subnet for that network, specified with the parameter <–subnet> ;
- Followed by the custom isolated network name.

To list all the networks, use the <docker network ls> command.

3. Embedded Domain Name Systems :

   It is possible to use the internal IP addresses to connect containers that are running at the same node. However, IP addresses can change when a system is rebooted : the proper way to do it is by using the container's name. Indeed Docker has a built-in DNS, Domain Name System, that helps the containers to resolve each other using their names. The DNS server always runs at the adresse 127.0.0.11.

4. Network Namespaces :

   Docker uses network namespaces that creates a separate namespace for each container in order to isolate each container within the host. The namespace then uses virtual Ethernet pairs to connect containers together.

VI. Docker Storage :

1. File Systems :

When Docker is installed on a system, it creates the following folder at /var/lib/docker. This is where Docker stores all its data by default. This folder contains many other folders such as aufs, containers, image, volumes, et

2. Volumes :

Due to layered architecture, images and containers have a copy-on-write mechanism. Containers are stored in read-write files and can be modified. Images, on the other hand, are stored in read-only files, which means that the files in these layers will not be modified in the image itself : the image remains the same at all times until it is rebuilt. Upon rebuild, anything depending on that image before the rebuild including the read-only files storing the images are deleted.

Volumes can persist any data dependent on the image by adding, what is called a "persistent volume", to the container. The command <docker volume create> <file name> will create a specified folder in the pre-existing volumes folder. By now, it is possible to run a container and mount the volume inside its read-write layer adding the option <-v> <volume name>:<path to container> to the <docker run> command.

You can also use the option <-mount> instead of the <-v> option. However, the parameters will be under the following format : <docker run > <-mount> <volume name>=<path to container>.

In short, there are two types of mounts :
- Volume mounting mounts a volume from the volumes directory ;
- Bind mounting mounts a directory from any location on the Docker host.

a. Volume Mounting :

Docker automatically creates a volume when running the <docker run> command with the <-v/-mount> option : the usage of the <docker volume create> is not obligatory. This is called volume mounting : we are mounting a volume created by Docker under the folder /var/lib/docker/volumes.

b. Bind Mounting :

In order to stock volume data in another folder that is pre-existing, use the <docker run> command, again, with the <-v/-mount> but by specifying a folder with it's path as first parameter : <docker run> <-v/-mount> <path to external folder>:/=.<path to container>. This is called bind mounting.

3. Storages Drivers :

Docker uses storage drivers that are responsible for any operation such as maintenance of the layered architecture, creating a writable layer, moving files across layers to enable copy and write, etc.

Some of the common storage drivers for layered architecture are AUFS, BTRFS, ZFS, Device Mapper, Overlay and Overlay2. The selection of the storage driver depends on the underlying OS being used. For example, the default storage drive for Ubuntu is ZFS whereas this storage driver is not available on other operating systems like Fedora (which uses Device Mapper). Anyway, Docker will choose the best storage driver available automatically based on the operating system. However, different storage drivers also provide different performances and stability characteristics, so perhaps the default storage drivers aren't always the best.

4. Docker Registry :

Docker images are stored by default in Docker's default registry : Docker Hub, which DNS is <docker.io>. It's a central repository of all docker images, this to say whenever a new one is created or if one is updated it is stored there. Then, when anyone deploys the application, it is pulled from that specific registry. However, many other public registries exist as well such as Google's registry which offers a lot of kubernetes related images : GCR. When it comes to private registries, we may find many cloud service providers such as AWS that provide a private registry by default when an account is opened with them.

Whatever the public registry used, it is possible to make a repository private so that it can only be accessed using a set of credentials. From Docker's perspective, to use an image from a private registry, it is required to first log into that private registry using the <docker login> command followed by your credentials. Once the login is successful, run the application by using the <docker run> command followed with <private-registry> as part of the image name. However, if you haven't logged into the private registry beforehand the command will come back saying the image cannot be found.

To deploy an application using private registries such as cloud providers on-premise without having a private registry is it required to deploy your own private registry within your organization. The Docker registry is itself another application and is, of course, available as a Docker image that is named <registry> and that exposes the API, Application Programming Interface, on port 5000. Therefore, to run your custom registry use the <docker run> command using the Docker's registry image that is <registry:2>. Specify a port using the 5000 host port. To push an image to that port <docker image tag> to tag the image with the private registry URL in it (starting with <localhost:500>). Then, push the image to the local private registry using the <docker push> and the new image name with the docker registry information in it. From there, the image can be pulled anywhere within the network using either <localhost:5000> if you're one the same host or by using the IP or domain name of the Docker host if it's accessed from another host.

5. <u>Docker Engine :</u>

Docker engine simply refers to a host with Docker installed on it. Apon installing Docker, you actually instal 3 different components :

- The Docker Daemon which is a background process that manages objects such as images, containers, volumes and networks ;
- The REST API server with is the API interface that programs can use to talk to the daemon and provide instructions (it is also possible to create your own tools using this API) ;
- The Docker CLI which is nothing but the command-line interface that is used to perform actions such as running a container, stopping a container, destroying images, etc. It uses the REST API to interact with the Docker Daemon. However, it doesn't necessarily need to be one the same host : it can be on another system such as a laptop and still work with a remote Docker engine by simply using the <-H=> option on the Docker command and specify the remote Docker engine address and a port.

a. Namespaces and IDs :

Docker uses namespaces to isolate workspace. Process IDs, network, inter-process communication, mounds and unix time sharing systems are created in their own namespace. Thereby providing isolation between containers.

One of many namespace isolation techniques is what is called process ID namespaces. By definition, whenever a Linux system boots up, it starts with just one process with a process ID of one also known as the root process. This process kicks off all the other processes in the system. By the time the system boots up completely, there is not only one process but a handful of processors running simustaneley (to see the running processes, run the ps command). Each process ID is unique, therefore two processes cannot share the same process ID.

To create a container, which is basically a child system within the current system, the child system needs to believe it is an independent system on its own with its own set of processes originating from a root process with a process ID of one. However, there is no hard isolation between the containers and the underlying host. Therefore, the processes running inside the container are in fact processes running on the underlying host meaning processes cannot have the same process ID of one. This is where namespace comes into play.

With process ID namespaces, each process can have multiple process IDs associated with it. For example, when processes start in the container, it is actually just another set of processes on the system that gets the next available process ID available. In other words, all processes are in fact running on the same host but separated into their own containers using namespaces.

b. Cgroups :

As a reminder, the underlying Docker host, as well as the containers, share the same system resources such as CPU, Central Processing Unit, and memory. Resources are therefore dedicated to the host and the containers, and shared once again by the different containers. Indeed, by default, there is no restriction as to how much of a resource a container can use and hence a container may end up utilizing all the resources on the underlying host.

To restrict the amount of CPU or memory a container can use, Docker uses Cgroups, also known as control groups, to restrict the amount of hardware resources allocated to each container. This can be done by providing the <--cpus=> option to the <docker run> command providing a value that will ensure that the container will not take up more than the specified amount of the host's CPU at any given time. The same goes for memory : <docker run –memory=> followed by the maximum amount of memory the specified container can use on the host.

VII. Docker Compose :

To set up a complex application running multiple services it is better to use Docker Compose instead of the <docker run> command. With Docker compose it is possible to create a configuration file in yamo format called <docker-compose.yml>. The file will be composed of all the different services and the options specificity to run them in that file. To then simply run a <docker compose up> command to bring up the entire application stack.

Docker Compose is a better option as it is easier to implement, run and maintain as all changes are always stored in the Docker Compose configuration file. However, this is all only applicable to running containers on a single Docker host.

1. From Docker Run To Docker Compose :

Instead of having to run (<docker run>) individually all the different containers of an application and linking (<--link>) them together manually a <docker-compose.yml> file can be used to simplify this process.

Here is a simple step by step example of how to construct a Docker Compose file. This example is based on a voting application that requires a voting app developed in Python, a Redis in-memory database, a Worker application written in Dotnet, another database which is a Postgres SQL and finally a web interface developed in node.js.

a. Step One :

Create a dictionary of key container names by using the same names used in the <docker run> commands ;

Docker Run	Docker Compose yml File
docker run -d --name=redis redis	redis:
docker run -d --name=db postgres:9.4	db:
docker run -d --name=vote 5000:80 --link redis:redis voting-app	vote:
docker run -d --name=result -p 5001:80 --link db:db result-app	result:
docker run -d --name=worker --link db:db --link redis:redis worker	worker :

b. Step Two :

Specify for each container which image to use by creating a property called <image> : the key is the image and the value is the name of the image to use.

Docker Run	Docker Compose yml File
docker run -d --name=redis redis	redis: image: redis
docker run -d --name=db postgres:9.4	db: image: postgres:9.4
docker run -d --name=vote 5000:80 --link redis:redis voting-app	vote: image: voting-app
docker run -d --name=result -p 5001:80 --link db:db result-app	result: image: result-app
docker run -d --name=worker --link db:db --link redis:redis worker	worker: image: voter

c. <u>Step Three :</u>

Inspect the commands and see what other options are used, such as ports by creating a property called <ports> and links, this time with the property <links>.

Docker Run	Docker Compose yml File
docker run -d --name=redis redis	redis: image: redis
docker run -d --name=db postgres:9.4	db: image: postgres:9.4

docker run -d --name=vote 5000:80 --link redis:redis voting-app	vote:
	    image: voting-app
	    ports:
	        - 5000:80
	    links:
	        - redis

docker run -d --name=result -p 5001:80 --link db:db result-app	result:
	    image: result-app
	    ports:
	        - 5001:80
	    links:
	        - db

docker run -d --name=worker --link db:db --link redis:redis worker	worker: image: worker links: - redis - db

d. Step Four :

Bring up the entire application stack with the <docker compose up> command. However, images can not be built and available in the Docker registry prior to this command. To do so, instruct Docker Compose to run a <docker build> an image with the property <build> instead of the <image> property that just pulls the specified image. The <build> property requires the location of the directory which contains the application code a Dockerfile with instructions to build the Docker image.

Docker Run	Docker Compose yml File
docker run -d --name=redis redis	redis: image: redis
docker run -d --name=db postgres:9.4	db: image: postgres:9.4
docker run -d --name=vote 5000:80 --link redis:redis voting-app	vote: build: ./vote ports: - 5000:80 links: - redis
docker run -d --name=result -p 5001:80 --link db:db result-app	result: build: ./result ports: - 5001:80 links: - db
docker run -d --name=worker --link db:db --link redis:redis worker	worker: build: ./worker links: - redis - db

2. Docker Compose Versions :

Docker Compose files can come in different formats at different places as Docker Compose evolved over time and now supports a lot more options than it did before. In order for Docker Compose to know which version of it is used, it is required to specify which version is used with the <version> property (necessary for version 2 and above).

a. Version 1 :

Docker Compose Version 1	

redis:
  image: redis
db:
  image: postgres:9.4
vote:
  image: voting-app
  ports:
    - 5000:80
  links:
    - redis
	Version 1 of Docker Compose is pretty limited as you can't deploy containers on a different network other than the default bridge network and you can't have a dependency or startup order.

b. Version 2 :

Docker Compose Version 2	

version: 2
services:
    redis:
        image: redis
    db:
        image: postgres:9.4
    vote:
        image: voting-app
        ports:
            - 5000:80
        depends_on:
            - redis
	However, the version 2 of Docker Compose offers these possibilities but its format changes slightly : you don't specify the stack's information directly, instead it is all encapsulated in a service section. To do so, create a property called <services> in the root of the file and then move all the underneath it.
	
	When it comes to the network of version 2, Docker Compose automatically dedicates a bridged network for the application and then attaches all containers to that network. The containers are then able to communicate with one another using each other's service name. Therefore, <links> properties are not required in version 2.
	
	Version 2 also introduces it <depends_on> property : you can specify a startup order with it.

c. Version 3:

Docker Compose Version 3	

version: 3
services:
    redis:
        image: redis
    db:
        image: postgres:9.4
    vote:
        image: voting-app
        ports:
            - 5000:80
        depends_on:
            - redis
	Version 3 is the latest version of Docker Compose and is similar to version 2 : it has the <version>, <depends_on> and <services> properties and also comes with support for Docker Swarm.

3. Docker Compose Network :

Docker Compose Version 2

```
version: 2
services:
   redis:
      image: redis
      networks:
         - back-end
   db:
      image: postgres:9.4
      networks:
         - back-end


   vote:
      image: voting-app
      networks:
         - frond-end
         - back-end

   result:
      image: result
      networks:
         - frond-end
         - back-end

network:
   front-end:
   back-end:
```

So far, Docker Compose deployed all containers on the default bridge Network. With version 2 and 3, Docker Compose automatically dedicates a bridged network for the application and then attaches all containers to that network. However, it is also possible to create multiple networks inside an application.

Define the networks that will be used with the property <networks> at the root level adjacent to the <services> property and add a map of networks that are going to be used. Then under each service add another property a <network> property and provide a list of networks that service must be attached to.

VIII. Docker on Different Operating Systems Kernel :

As a reminder, containers share the same underlying OS Kernel. As a result, it is not possible to have a Windows containers running on a Linux host or vice versa.

1. Docker on Windows :

To use Docker on Windows, there are two available options. The first being to use Docker on Windows using Docker Toolbox, and the second being Docker Desktop for Windows.

a. Docker Toolbox :

Docker Toolbox was the original support for Docker on Windows. Indeed, is it required to install a virtualization software on the Windows system such as Oracle, VirtualBox or VMware Workstation, and deploy a Linux VM on it like Ubunton or Debian. Then, install Docker on the Linux virtual machine and then play with the endless possibilities that Docker offers.

This option doesn’t really have anything much to do with Windows : it is not possible to create Windows based Docker images or containers. Linux or other OS Kernel can't be run either. This option only offers to work with a Docker on Linux VM on a Windows host.

However, Docker provides a set of tools to make this easy which is known as the Docker Toolbox. This set of tools includes Oracle Virtualbox, Docker engine, Docker machine, Docker compose, and a user interface called Kitematic GUI, a lightweight VM called boot to Docker which has Docker running already on it, etc. To use this Docker Toolbox, the operating system has to be of 64-bits, the Windows version must be 7 or higher and the virtualization must be enabled on the system.

Note that Docker toolbox is a legacy solution for older Windows systems that do not meet the requirements for the newer Docker for Windows option.

b. Docker Desktop for Windows :

Docker Desktop for Windows takes out Oracle Virtualbox and uses the native virtualization technology available with Windows called Microsoft Hyper-V. During its installation process, it will still automatically creates a Linux underneath. However, this time, it is created on the Microsoft Hyper-V instead of Oracle Virtualbox and has Docker running on that system.

Due to the dependency on Hyper-V, this option is only supported for Windows 10 Enterprise or Professional Edition and on Windows Server 2016 as both of these operating systems come with Hyper-V support by default. However, Docker support for Windows is strictly for Linux containers and Linux applications packaged into Linux Docker images.

c. Windows Containers :

With Windows Server 2016, Microsoft announced support for Windows containers for the first time. Therefore, it is now possible to package Windows applications into Docker Windows containers and run them on Windows Docker host using Docker Desktop for Windows.

When installing Docker Desktop for Windows, the default option is to work with Linux containers even if it is possible to use Windows containers. However, to use Windows containers, it is required to explicitly configure Docker for Windows to switch to using Windows containers.

Unlike on Linux, there are two types of containers on Windows :

- Windows Server containers works exactly like Linux containers where the OS Kernel is shared with the underlying OS ;
- Hyper-V Isolation containers allow better security boundaries between containers and to allow Kernels with different versions and configurations to coexis (each container is run within a highly optimized VM guaranteeing complete Kernel isolation between the containers and the underlying host).

2. Docker on MacOS :

   Docker on Mac is similar to Docker on Windows. There are two options to get started : the first one being Docker on Mac using Docker toolbox, and the second being Docker Desktop for Mac.

   a. Docker toolbox :

      Docker toolbox was the original support for Docker on Mac. It is, once again, Docker on Linux VM created using Virtualbox on Mac. As with Windows, it has nothing to do with Mac applications, Mac based images of Mac containers : it purely runs Linux containers on a Mac OS. Once again, Docker toolbox contains a set of tools such as Oracle Virtualbox, Docker engine, Docker machine, Docker Compose, a user interface called Kitematic GUI, etc. Docker toolbox requires a MacOS 10.8, also known as the "Mountain Lion" version, or newer.

   b. Docker Desktop for Mac :

      Docker Desktop for Mac takes out a commercial box and uses HyperKit virtualization technology. During its installation process, Docker for Mac will still automatically create a Linux System underneath but this time it is created on HyperKit instead Oracle Virtualbox. This requires MacOS Sierra 10.12 or newer and the Mac hardware must be a 2010 model or newer. For now, there are no Mac based images or containers : all of this is just to be able to run Linux containers on Mac.

IX. Docker Orchestrate :

1. Problem - Why use a container orchestration :

So far, Docker can run a single instance of an application with a simple <docker run> command. If the number of users increases and that instance is no longer able to handle the load, you simply run additional instances of the application by running the <docker run> command as many times as needed. This multiple run of containers is done manually, it requires you to do it by having to keep a close watch on the load and performance of the application and deploy additional instances yourself. You also have to keep a close watch on the application's health : if the application crashes, you need to be able to detect it and run the <docker run> command again to deploy another instance of the application. However, a host can also crash and no longer be accessible.

2. Solution - What is a container orchestration :

To prevent a constant watch of the application, it is possible to write scripts that will tackle the issues listed beforehand. This solution is known as container orchestration : it is a solution that consists of a set of tools and scripts that can help host containers in a production environment. Typically, a container orchestration solution consists of multiple Docker hosts that can host containers, that way, even if one fails the application is still accessible through the others.

A container orchestration solution allows you to deploy hundreds or thousands of instances of an application with a single command. Some solutions can help automatically scale up the number of instances when users increase and scale down the number of instances when the demand decreases. Other solutions can even help in automatically adding additional hosts to support the user load. Moreover, container orchestration solutions can also provide support for advanced networking between these containers across different hosts as well as load balancing user requests across different hosts. They also provide support for sharing storage between the hosts, support for configuration management and security within the cluster. And the goes on and on.

3. Different Types of Container Orchestration :

   Many orchestration solutions are available today such as Docker Swarm from Docker, Kubernetes form Google, and Mesos from Paget. While Docker Swarm is really easy to set up and get to speed with, it lacks some of the advanced auto scaling features required for complex production grade applications. Mesos, on the other hand, is quite difficult to set up and get started but supports many advanced features. Finally, Kubernetes is arguably the most popular solution of them all : it is a bit difficult to set up and get acquainted with, but provides many options to customize deployments and has support for a lot of different vendors. Kubernetes is now not only supported on all public cloud service providers such as GCP Azure and AWS, it is also one of the top-ranked projects on Github.

4. Docker Swarm - Docker :

   In short, with Docker Swarm it is now possible to combine multiple Docker machines together into a single cluster. Docker Swarm will take care of distributing the services or application instances into separate hosts for high availability and for load balancing across different systems and hardware.

   Setting up a Docker Swarm requires having one or multiple hosts with Docker installed on them. Then, designate one host to be the "manager", also called the "master" or the "swamp" manager, while the other will remain as "slaves" or "workers". Once done, run the <docker swarm init> command on the swarn manager and that will initialize the swamp manager. The output will also provide the command to be run on the workers : copy the command and run it on the workers nodes to join the manager. After joining the swamp, Docker Swamp is now ready to create services and deploy them on the swamp cluster.

   a. Swamp Cluster :

      There are two ways to create a swamp cluster and use it to run multiple instances of the server. First of all, it is possible to do so by running the <docker run> command on each worker node. However, this is not ideal as you may need to log into each node individually and run the command. Depending on the amount of nodes, you could possibly have to do so for hundreds of nodes to set up : it could end up being an impossible task. This is where Docker Swarm orchestration comes in handy : Docker Swarm orchestrator does all of this for us.

b. Swarm Services and Stacks :

A Swarm is a group of machines that run the Docker engine and that are part of the same cluster. Docker Swarm allows you to launch Docker commands that are usually used on a cluster from a master machine named manager/leader Swarm. Machines that join a Swarm are called nodes.

Docker Stacks is run across a Docker Swarm. Therefore Stacks are grouped machines running on a Docker daemon that essentially pools resources. Stacks allow for multiple services : these services are containers distributed across a swarm that are deployed and grouped logically.

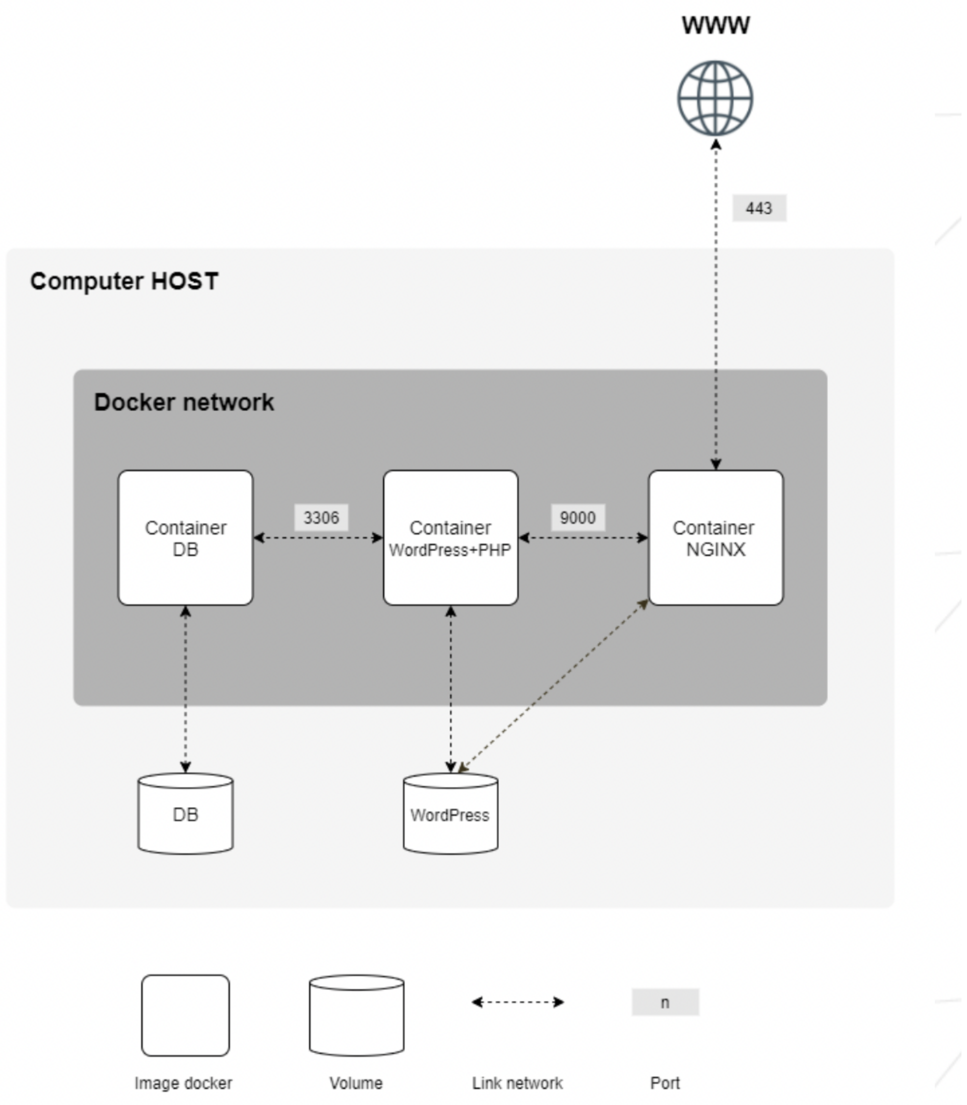
Docker services are one or more instances of a single application or service that runs across the nodes in the swamp cluster. To do so, use the <docker service create> command, specify the image name and add the option <--replicase=> to specify the number of instances to run across the cluster and that will be distributed across the different worker nodes. Note that the <docker service> command must be run on the manager and not on one of the worker nodes. The <docker service create> command is similar to the <docker run> command in terms of options passed such as the <-e> environment variable, the <-key> for publishing ports, the network option to attach containers to a network, etc.

X. Project :

1. Subject :

Set up as followed :

- A Docker container containing NGINX with TLSv1.2/TLSv1.3 only ;
- A Docker container containing WordPress and php-fpm (installed and configured) only without nginx ;
- A Docker container containing MariaDB only without nginx ;
- A volume containing your WordPress database ;
- A second volume containing your WordPress site files ;
- A docker-network that will link your containers.



2. Setting up MariaDB :

   a. What is MariaDB ?

      MariaDB is a :
      - Community-developed commercially supported fork (drop-in replacement) of MySQL ;
      - Relational database management system, intended to remain free and open-source software under the GNU General Public License.

Advantages
Offers tighter security measures which is a major concern for any website owner.
Performance is faster and more efficient.
Access to better user support.
Allows versioned tables creating as it overall faster and easier-to-use than other databases.

   b. MariaDB Storage System :

      When it comes to databases, you usually have to have different orchestrations, backup methodologies, application layers, etc : all this causes high operational overhead.

      However, MariaDB offers all the aspects listed beforehand with the same sequel layer on top and different storage engines underneath within one binary.

   c.

3.

XI.