## SECTION 03

*Containerization with Docker*

## WHAT IS DOCKER?

- It is a way of running multiple software applications on the same machine. Each of which is run in an isolated environment called “container”.
- It eliminates the annoying problem of “works on my machine”. Software is packaged in the container and, hence, can be run on any machine that has the Docker engine installed.
- A container is a closed environment for the software. It bundles all the files and libraries that the application needs to function correctly. Multiple containers can be deployed on the same machine and share the resources.
- But isn’t that what virtual machines and Vagrant do?
- No, virtualization is the process of abstracting the operating system hardware (CPU, memory, network… etc.) so that complete machines can be run simultaneously on the same host. Containers - on the other hand - uses the same physical hardware of the host machine to run just the application in an isolated environment.
- Containers - by definition - use less resources than virtual machines. however, they have the drawback of using Linux kernels only. You cannot containerize a Windows or a MAC application.

## HOW DOES IT WORK?

- You learned that Vagrant uses boxes to distribute virtual machines. A box can be used to spawn multiple machines. Docker works in the same manner. It uses images to spawn containers. Multiple containers can be created using an image.
- Images are indexed and saved in an online repository managed by Docker. It can be found on https://hub.docker.com/explore/. Of course you can also create and maintain your own images.
- To understand the power of Docker consider the following example:
  - You have a web application running on Ubuntu 15.
  - You need to upgrade your OS to the latest Ubuntu 17. But then the software will not run as it depends on libraries that are only available in Ubuntu 15.
  - Using a Docker image to run this software, you can upgrade Ubuntu to the latest version and ensure that the application will continue to run in complete isolation of the OS current libraries.

## INSTALLATION AND FIRST EXAMPLE

- Docker can be installed on Windows, Linux, and MAC. But it can only run applications designed to work on a Linux kernel.
- Currently, Docker supports two flavors: the community edition (CE) and the enterprise edition (EE). The CE is free of charge but with community support. The EE is a paid version but you have enterprise-level support from Docker.
- The following pages lists the different versions you can use and the download links https://docs.docker.com/engine/installation/#desktop
- On a Windows or a MAC machine, just download the installer and follow the wizard. On Linux, it will take some steps. We're using an Ubuntu 17 OS for this demonstration.
- Follow these steps to install Docker CE on Ubuntu:

- `sudo apt-get update`
- `sudo apt-get install apt-transport-https ca-certificates curl software-properties-common`
- `curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -`
- `sudo apt-key fingerprint 0EBFCD88`
- `sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"`
- `sudo apt-get update`
- `sudo apt-get install docker-ce`
- To verify your installation, run the following command: `docker run hello-world`

## THE DOCKERFILE

- Like the Vagrantfile for Vagrant machines, the Dockerfile is an instructions file used for building containers.
- However, Vagrant *must* have a Vagrantfile for the machine to work. Docker needs the Dockerfile only if you are building new images.
- Building an image is an incremental process. For example, you may download the Apache web server image (httpd), and use a Dockerfile to install components on top of it like a specific version of PHP.
- You can add new packages, change configuration files, create new users and groups, and even copy files and directories to the image.
- An example of a Dockerfile can be examined through this URL: https://github.com/docker-library/httpd/blob/master/2.4/Dockerfile This is the Dockerfile that was used to create the official Apache image, called httpd.

## BASIC DOCKER COMMANDS

- All commands that deal with Docker containers and images start with `docker` command followed by subcommands and switches. You can use `docker --help` or `docker` *`subcommand`* `--help`. For example: `docker images --help`
- Let's pull an image, called busybox from the repository. Busybox is a tiny Linux kernel that has very basic functionality. Issue the following command: `docker run --detach --name server busybox`
- A combination of numbers and letters appears (called container id) and that's it. But actually the container has already run and stopped. Remember, docker containers are just processes. If you run a process that does nothing, it will just exit the moment it runs.
- You can start, stop, restart a container using `docker start/stop/restart` *`container`*
- You can remove a container using docker rm container
- You can connect to the container's shell using `docker attach container`. Notice that the container is stopped as soon as you exit the shell as this is the process that keeps it running.

- If you want to see the status of the current containers (docker processes) run docker ps. In our case, nothing is returned.
- Let's make it do something that will prevent it from exiting: `docker run --detach --name server busybox sh -c "while true; do sleep 5; done"`
- You can add any shell command after the docker run command and the container will run it. In this case, we instructed the container to run an infinite loop that will sleep for five seconds every iteration.
- Now if you run `docker ps`, you will find the container process listed.
- The `--detach` (or `-d`) makes the container run in the background, just like adding the ampersand & at the end of a command to make the process run in the back.
- You can have the container running in the foreground if you want to see its output, possibly any errors or notifications.

- In this case, we use `-i` (for interactive). It is recommended that you also use the `-t` (TTY) switch, which will instruct the container to accept shell signals like normal processes. Otherwise, you won't be able to exit the container running in the foreground by pressing CTRL-c for example.
- Let’s run our container in the foreground and make it do something more useful:
  ```
  docker run -it --name iserver busybox sh -c "while true; do
  date; sleep 5;done"
  ```
- The above command will make the container print the current date and time every five seconds. The output is printed to the screen and you cannot type any commands to the shell until you exit the container by pressing CTRL-c.
- Notice that we used a different name for the container, iserver. If you tried to use the same name for this container you will receive an error that the container with the name “server” is already in use. You’d had to remove it first by running `docker rm server`

## DOCKER NETWORKING

- A Docker container is not a virtual machine. This means that it uses the existing network interface of the host.
- Once a container is created, a private loopback interface is available for the container for internal communication. It is totally isolated from the outside traffic.
- Docker also creates a bridge interface called docker0 that is responsible for carrying traffic from and to the container.
- The docker0 interface acts as a router for the container. It can be used for internal communications between different containers and the host. It can also be used for routing traffic from and to the outside network.
- Docker provides three modes of networking: closed, bridged and open.

## CLOSED CONTAINERS

- In this scenario, the container is totally isolated from any outside networks. It cannot communicate even with the host.
- You can use model when you want to seal the container from any traffic.
- The loopback interface is used only for internal communication inside the container.
- To create a closed container you use the --net switch and supply none for an argument. Example: `docker run -it --rm --net none busybox ifconfig`
- The above command will create a new container based on the busybox image, but with not network interfaces. The output of ifconfig run on the container lists only the lo adapter.
- You may have noticed the use of --rm switch. This will ensure that the container gets deleted as soon as it finishes running. This will save you from having a lot of unused containers.

## BRIDGED CONTAINERS

- This is the default network model for Docker containers. Running `docker -it --rm busybox ifconfig` will list a second interface eth0 in addition to the loopback one. It has an IP address that was supplied by the docker0 bridge interface.
- You can also specify that the container needs to be configured in the bridged mode by specifying `--net bridge`. However, this is redundant as bridge is the default mode.
- The bridge interface allows containers to access the outside network and the Internet through the host. This can be used to download important OS updates for example. But no inbound connections are allowed unless specified. This provides a level of security to the container.
- Let's see how we can access the container’s internal ports.

## COMMUNICATING WITH THE CONTAINER

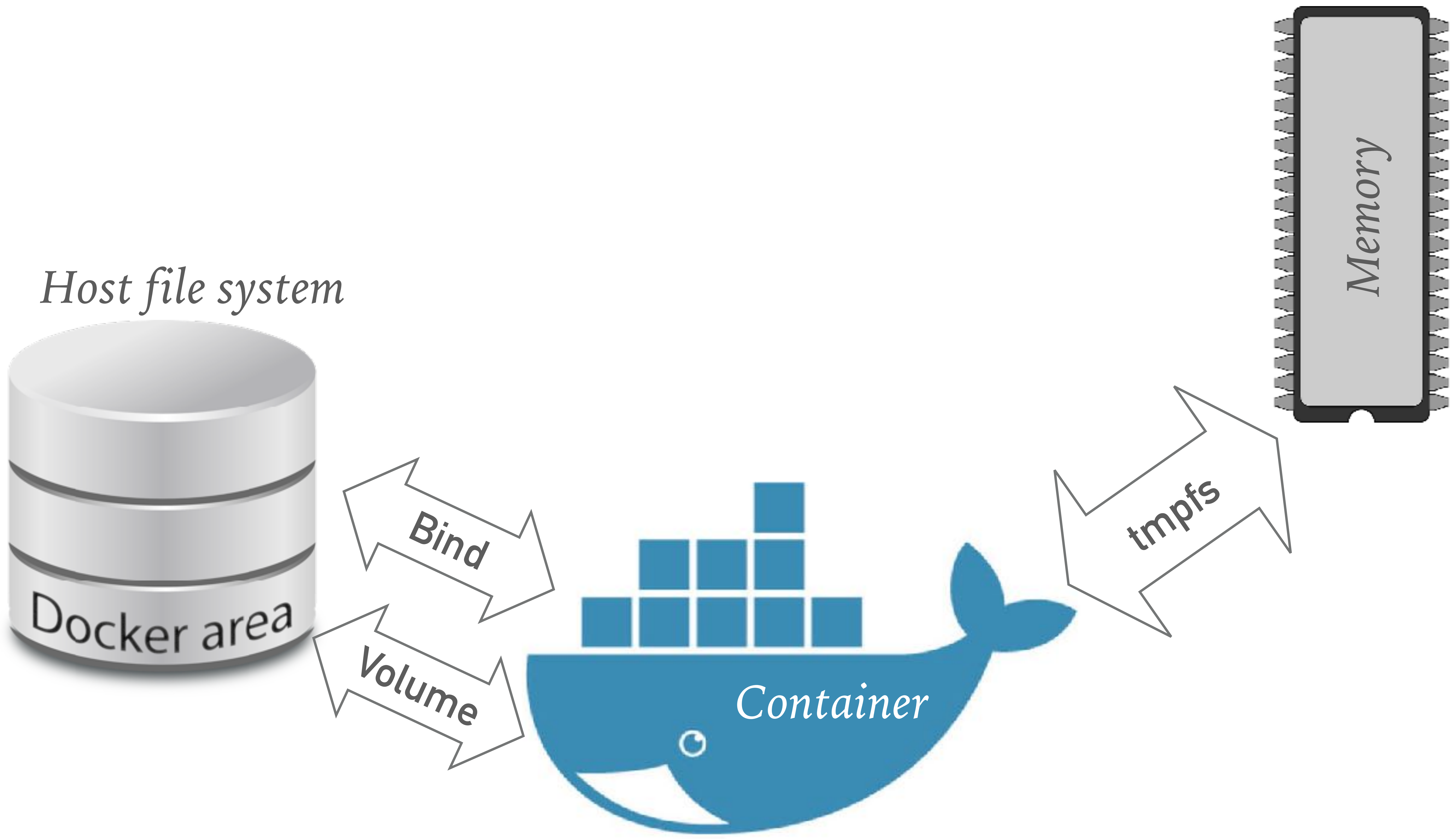
- When creating the container, you have four options to access its internal network:
  - Forwarding traffic to the port through a random port on the host. For example: `docker run -d -p 80 --name webserver httpd`
  - You can determine the random port that was chosen by running `docker ps`
  - You can also bind this random port to a specific IP address on the host (if you have multiple interfaces/addresses) by modifying the command to be: `docker run -d -p 127.0.0.1::80 --name webserver httpd`. This will accept traffic on the host from the same machine only.
  - Forwarding traffic to the port through a specified port on the host. This is the most commonly uses scenario. For example: docker run -d --name webserver -p 8080:80. Now all traffic coming at port 8080 on the host will get forwarded to port 80 on the container.
  - The same option applies here for binding the port to a specific address by running the following command: `docker run -d --name webserver -p 127.0.0.1:80:8080` to make the host accept forwarded traffic only on the loopback interface.

## OPEN CONTAINERS

- Open containers have full access to the host’s network. No port forwarding is in place.
- Even ports lower than 1024 (reserved ports) are directly accessible from the host.
- You'll have to be very cautious when using this model as it provides the least security.
- For example: `docker run -d --name webserver --net host httpd`
- Now navigating to the localhost on the host browser will bring the default page of the Apache webserver.
- As mentioned, the bridged mode is the default one used by containers when they are first created. It is used for intercommunication between the containers as we’ll see later in this section.

## DOCKER STORAGE

- Docker uses UFS (Union File System) to work. This basically means that an image may be a combination of a number of images, each of which has its own file systems. Those file systems are *layered* together so that common files and libraries can be reused.
- This filesystem is not suitable for all scenarios. When you want to share files with a Docker container, you'd use volumes for this.
- A volume is just a mount point to a directory inside the container where files can be shared with the host. It works just like /vagrant that we covered earlier in the Vagrant section, but of course in a technically different manner.
- Actually, Docker supports three methods of storage: volume, bind mounts, and tmpfs mounts.


Host file system
Docker area
Bind
Volume
Container
tmpfs
Memory

## BIND MOUNTS

- A bind mount refers to a mount point on the host that targets some destination directory on the container.
- The following example will create a web server container based on the httpd image. It will bind-mount the current directory on the host to the web directory of the container so that we can add HTML files: `docker run -d --name webserver -p 8080:80 -v "$(pwd)":/usr/local/apache2/htdocs/ httpd`
- The -v was used to specify the mount point on the host followed by the target directory on the container, separated by a colon.
- At any time you can view the bind mounts used by any container (in addition to the network configuration and other metadata about the container) by issuing `docker inspect` *`container_name`*

## DOCKER VOLUMES

- A Docker volume is a little more complex in setup than bind mounts. However, it is more robust and recommended to be used to save persistent data.
- To use a Docker volume with a container you can create one first. Let’s create a directory for our web server: `docker volume create web-data-vol`
- Now this is part of the host filesystem managed by Docker. To know where the volume was mounted just use `docker volume inspect web-data-vol`
- Actually you could’ve skipped this step as Docker will automatically create a volume for you once you start using it with the container if it is not already created.
- To use this volume with a container, you issue the same command used with the bind-mounts with a slight difference: `docker run -d --name webserver -p 8080:80 -v web-data-vol:/usr/local/apache2/htdocs/ httpd`
- The reasons why volumes are better than bind-mounts are numerous. The most important of which is that a name volume makes the container portable as it is not dependent on the host OS. In other words, you cannot refer to a bind mount in a Dockerfile, you can, however, refer to a named volume.

## THE TMPFS MOUNTS

- The tmpfs type of mount is used in scenarios where data does not need to be persisted on the host machine.
- An example of this scenario is when you need to generate a password or a hash that will be used in the current session only and does not (and should not) be available once the session is over.
- If you used a tmpfs mount point with a container, any data written to this location is going to be stored in the host's memory rather than the file system. Once the container is stopped, this data is gone.
- Notice that this feature does not work on Windows machines.
- For example, let's create a busybox container that will write a one-time hash to /tmp directory on the host machine: `docker run -itd --name server --tmpfs /tmp busybox`
- Login to the container and try to write some text to a file in /tmp. Once the container is restarted, this file is gone. Try to run the container without the --tmpfs and observe how the file persists during restarts.

## LAB: CREATING A LAMP STACK WITH WORDPRESS ON TOP OF IT

- In this lab, we are going to deploy the famous LAMP stack (Linux Apache MySQL PHP), and we will also deploy the well-know CMS WordPress on top of it using Docker.
- Let’s start by the database. We are going to use MariaDB (a forked version of MySQL) for this: `docker run -e MYSQL_ROOT_PASSWORD=admin -e MYSQL_USER=wordpress -e MYSQL_PASSWORD=wordpress -e MYSQL_DATABASE=wp_database -v wp_database:/var/lib/mysql --name wordpressdb -d mariadb`
- Notice here the use the the -e option. This allows you to inject environment variables into the containers. Environment variables are an excellent way to configure the container without having to log in and edit configuration files. In our case, the environment variables we are using set the root password, the MySQL user, the MySQL user password, the IP address of the host and the database name to be created.

- You may need to ensure that the database has been successfully installed and ready for WordPress. To do that you need to try connecting to it using a MySQL client.
- To install the client issue the following command: `apt-get install mysql-client`
- Now we need to get the IP address of the container. Using `docker inspect wordpressdb` you can look at the network settings part and get the IP address.
- The following command will login to MySQL: `mysql -u wordpress -p -h` *`ip_address`*
- You will be prompted for a password, enter wordpress. Once inside, you can verify that you have a wp_database created by typing `use wp_database;`
- The database container is ready, let's deploy the WordPress one. The following command will do just that: `docker run -e WORDPRESS_DB_USER=wordpress -e WORDPRESS_DB_PASSWORD=wordpress -e WORDPRESS_DB_NAME=wp_database -e WORDPRESS_DB_HOST=172.17.0.3  -p 8080:80 -v wp_data:/var/www/html  --name wordpressapp -d wordpress`
- Notice there the use of --link flag here.

## VAGRANT OR DOCKER?

- This is like asking whether to drive your car to work or take the bus. This highly depends on your use case and scenario.
- Vagrant is a virtualization software that works on top of a platform like VirtualBox or VMware. Using Vagrant, you have a complete virtual machine with it’s own CPU, memory, and network stack.
- Virtualization tends to consume a lot of resources since everything needs to be abstracted. However, it can be used in cloud platforms, for example, for instant provisioning.
- Docker uses containerization, which means that the image will create containers that share the same Linux kernel (the host), with the appropriate level of isolation. This means a lot less resource overhead. But at the end, it is just a Linux process running an application not a complete machine.
- To answer the question, actually they both complement each other with Vagrant abstracting the machine and Docker abstracting the application. For that reason, Vagrant uses Docker as one of its provisioners. This means that you can deploy a Vagrant machine that will automatically deploy a Docker container once it boots.