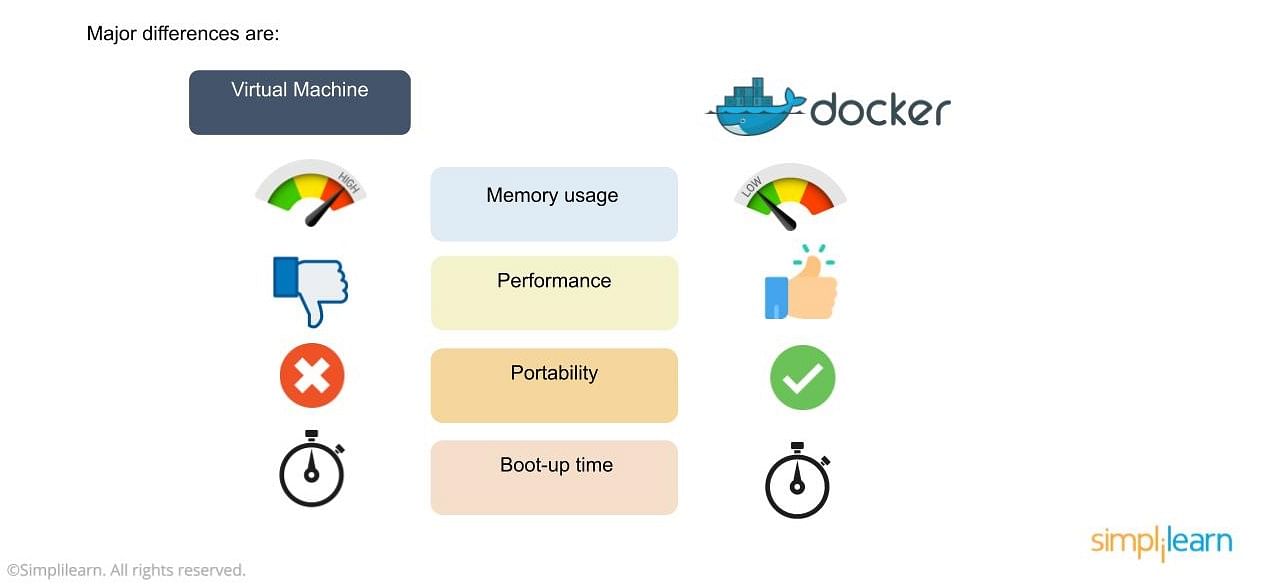
**What is Docker**

When getting started with Docker, we first need to understand Docker. Docker is an OS virtualized software platform that allows IT organizations to quickly create, deploy, and run applications in Docker containers, which have all the dependencies within them. The container itself is a very lightweight package with all the instructions and dependencies—such as frameworks, libraries, and bins—within it.

The Docker container can be moved from environment to environment very easily. In a [DevOps life cycle](https://www.simplilearn.com/steps-to-building-momentum-for-devops-article), Docker really shines when used for deployment. When you deploy your solution, you want to guarantee that the code tested will actually work in the production environment. In addition, when you're building and testing the code, it's beneficial to have a container running the solution at those stages because you can validate your work in the same environment used for production.

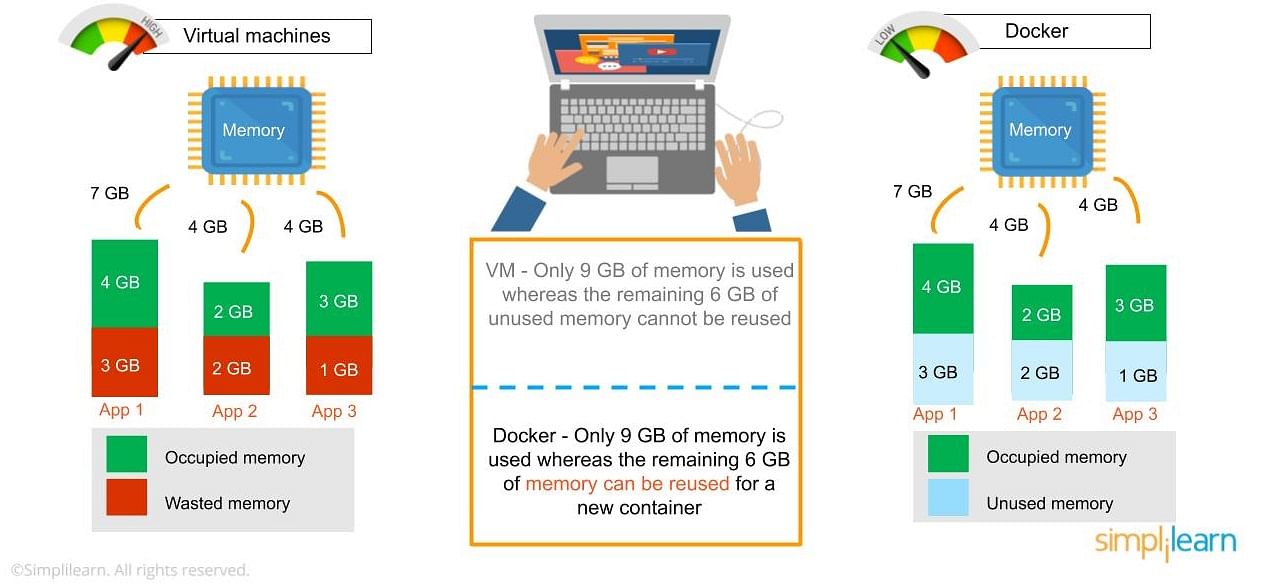
You can use Docker throughout multiple stages of your DevOps cycle, but it is especially valuable in the deployment stage, especially since it allows developers to use rapid deployment. In addition, the environment itself is highly portable and was designed with efficiencies that will enable you to run multiple Docker containers in a single environment, unlike traditional virtual machine environments.

Docker vs Virtual Machines



In the image, you’ll notice some major differences, including:

* The virtual environment has a hypervisor layer, whereas Docker has a Docker engine layer.
* There are additional layers of libraries within the virtual machine, each of which compounds and creates very significant differences between a Docker environment and a virtual machine environment.
* With a virtual machine, the memory usage is very high, whereas, in a Docker environment, memory usage is very low.
* In terms of performance, when you start building out a virtual machine, particularly when you have more than one virtual machine on a server, the performance becomes poorer. With Docker, the performance is always high because of the single Docker engine.
* In terms of portability, virtual machines just are not ideal. They’re still dependent on the host operating system, and a lot of problems can happen when you use virtual machines for portability. In contrast, Docker was designed for portability. You can actually build solutions in a Docker container, and the solution is guaranteed to work as you have built it no matter where it’s hosted.
* The boot-up time for a virtual machine is fairly slow in comparison to the boot-up time for a Docker environment, in which boot-up is almost instantaneous.



* One of the other challenges of using a virtual machine is that if you have unused memory within the environment, you cannot reallocate it. If you set up an environment that has 9 gigabytes of memory, and 6 of those gigabytes are free, you cannot do anything with that unused memory. With Docker, if you have free memory, you can reallocate and reuse it across other containers used within the Docker environment.
* Another challenge of virtual machines is that running multiples of them in a single environment can lead to instability and performance issues. Docker, on the other hand, is designed to run multiple containers in the same environment—it actually gets better with more containers run in that hosted single Docker engine.
* Virtual machines have portability issues; the software can work on one machine, but if you move that virtual machine to another machine, suddenly some of the software won’t work, because some dependencies will not be inherited correctly. Docker is designed to be able to run across multiple environments and to be deployed easily across systems.
* The boot-up time for a virtual machine is about a few minutes, in contrast to the milliseconds it takes for a Docker environment to boot up.

Advantages of Docker

Next in the getting started with docker tutorial we focus on the advantages of Docker. As noted previously, you can do rapid deployment using Docker. The environment itself is highly portable and was designed with efficiencies that allow you to run multiple Docker containers in a single environment, unlike traditional virtual machine environments.

The configuration itself can be scripted through a language called YAML, which allows you to describe the Docker environment you want to create. This, in turn, allows you to scale your environment quickly. But probably the most critical advantage these days is security.

You have to ensure that the environment you’re running is highly secure yet highly scalable, and Docker takes security very seriously. You’ll see it as one of the key components of the agile architecture of the system you’re implementing.

Now that you know the advantages of Docker, the next thing you need to know in this getting started with docker tutorial is how it works and its components.

How Does Docker Work?

Docker works via a Docker engine that is composed of two key elements: a server and a client; and the communication between the two is via REST API. The server communicates the instructions to the client. On older Windows and Mac systems, you can take advantage of the Docker toolbox, which allows you to control the Docker engine using Compose and Kitematic.

Now that we have learned about Docker, it's advantages, and how it works, our next focus in this getting started with docker tutorial is to learn the various components of Docker.

Components of Docker

There are four components that we will discuss in this getting started with docker tutorial:

* Docker client and server
* Docker image
* Docker registry
* Docker container

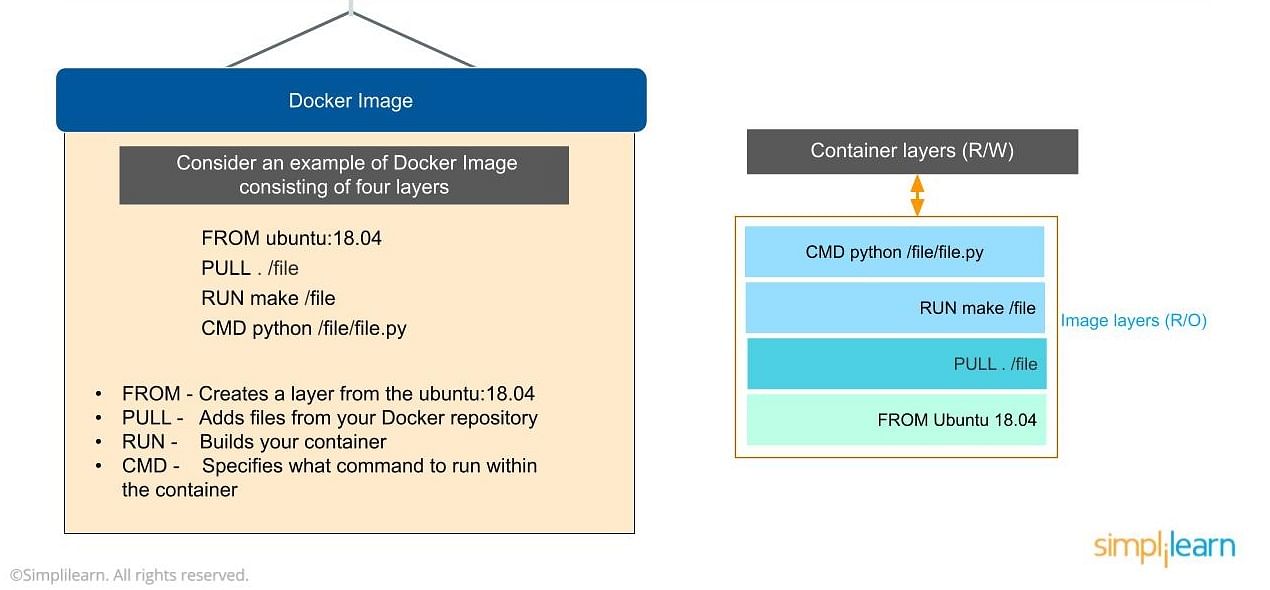
Docker Client and Server

This is a command-line-instructed solution in which you would use the terminal on your Mac or Linux system to issue commands from the Docker client to the Docker daemon. The communication between the Docker client and the Docker host is via a REST API. You can issue similar commands, such as a Docker Pull command, which would send an instruction to the daemon and perform the operation by interacting with other components (image, container, registry). The Docker daemon itself is actually a server that interacts with the operating system and performs services. As you’d imagine, the Docker daemon constantly listens across the REST API to see if it needs to perform any specific requests. If you want to trigger and start the whole process, you’ll need to use the Dockered command within your Docker daemon, which will start all of your performances. Then you have a Docker host, which lets you run the Docker daemon and registry.

Now let’s talk about the actual structure of a Docker image in this getting started with docker tutorial. A Docker image is a template that contains instructions for the Docker container. That template is written in a language called YAML, which stands for Yet Another Markup Language.

Docker Image

Next in the getting started with docker tutorial, we will learn all about Docker Image. The Docker image is built within the YAML file and then hosted as a file in the Docker registry. The image has several key layers, and each layer depends on the layer below it. Image layers are created by executing each command in the Dockerfile and are in the read-only format. You start with your base layer, which will typically have your base image and your base operating system, and then you will have a layer of dependencies above that. These then comprise the instructions in a read-only file that would become your Dockerfile.



Here we have four layers of instructions: From, Pull, Run and CMD. What does it actually look like? The From command creates a layer based on Ubuntu, and then we add files from the Docker repository to the base command of that base layer.

* Pull: Adds files from your Docker repository
* Run: Builds your container
* CMD: Specifies which command to run within the container

In this instance, the command is to run Python. One of the things that will happen as we set up multiple containers is that each new container adds a new layer with new images within the Docker environment. Each container is completely separate from the other containers within the Docker environment, so you can create your own separate read-write instructions within each layer. What’s interesting is that if you delete a layer, the layer above it will also get deleted.

What happens when you pull in a layer but something changes in the core image? Interestingly, the main image itself cannot be modified. Once you’ve copied the image, you can modify it locally. You can never modify the actual base image.

Docker Registry

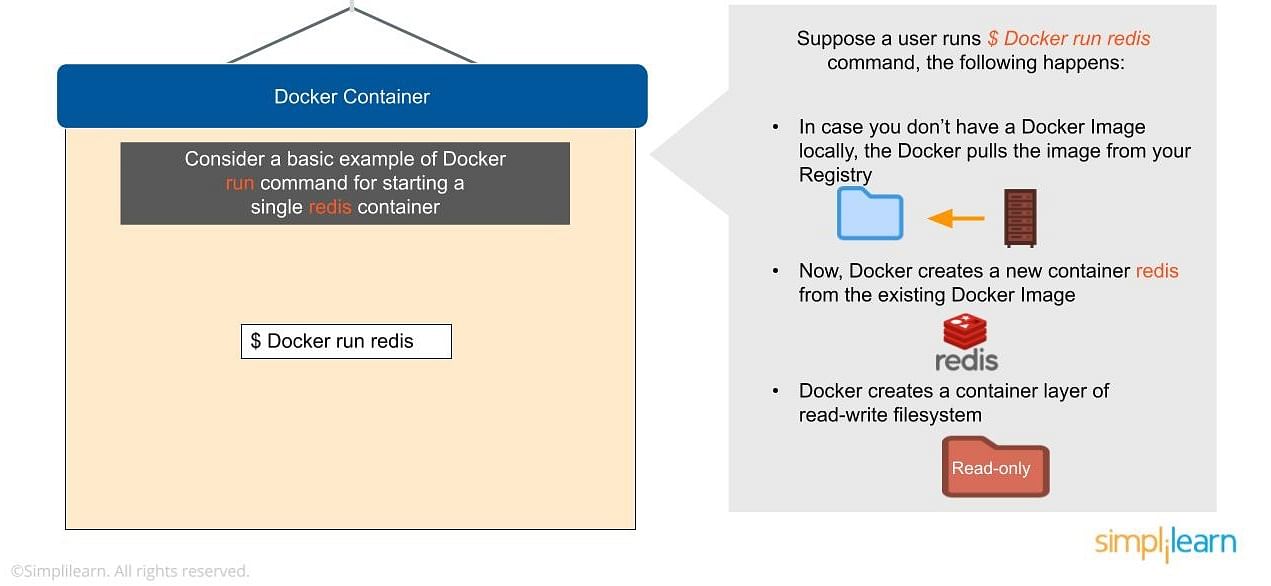
Next in the getting started with docker tutorial, we will learn all about Docker Registry. The Docker registry is where you would host various types of images and where you would distribute the images from. The repository itself is just a collection of Docker images, which are built on instructions written in YAML and are very easily stored and shared. You can give the Docker images name tags so that it’s easy for people to find and share them within the Docker registry. One way to start managing a registry is to use the publicly accessible Docker hub registry, which is available to anybody. You can also create your own registry for your own use internally.

The registry that you create internally can have both public and private images that you create. The commands you would use to connect the registry are  Push and Pull. Use the Push command to push a new container environment you’ve created from your local manager node to the Docker registry, and use a PullL command to retrieve new clients (Docker image) created from the Docker registry. Again, a Pull command pulls and retrieves a Docker image from the Docker registry, and a Push command allows you to take a new command that you’ve created and push it to the registry, whether it’s Docker hub or your own private registry.

Docker Container

Next in the getting started with docker tutorial, we will learn all about Docker Container. The Docker container is an executable package of applications and its dependencies bundled together; it gives all the instructions for the solution you’re looking to run. It’s really lightweight due to the built-in structural redundancy. The container is also inherently portable. Another benefit is that it runs completely in isolation. Even if you are running a container, it’s guaranteed not to be impacted by any host OS securities or unique setups, unlike with a virtual machine or a non containerized environment. The memory for a Docker environment can be shared across multiple containers, which is really useful, especially when you have a virtual machine that has a defined amount of memory for each environment.

The container is built using Docker images, and the command to run those images is Run. Let’s go through the basic steps of running a Docker image in this getting started with docker tutorial.



Consider a basic example of Docker run command for starting a single container called redis:

$ Docker run redis

If you don’t have the Redis image locally installed, it will be pulled from the registry. After this, the new Docker container Redis will be available within your environment so you can start using it.

Now let’s look at why containers are so lightweight. It’s because they do not have some of the additional layers that virtual machines do. The biggest layer Docker doesn’t have is the hypervisor, and it doesn’t need to run on a host operating system.

Now that you know the basic Docker components, let’s now look into advanced Docker components in this getting started with docker tutorial.

Advanced Docker Components

After going through the various components of Docker, the next focus of this Docker tutorial are the advanced components of Docker:

* Docker compose
* Docker swamp

Docker Compose

Docker compose is designed for running multiple containers as a single service. It does so by running each container in isolation but allowing the containers to interact with one another. As noted earlier, you would write the compose environments using YAML.

So in what situations might you use Docker compose? An example would be if you are running an Apache server with a single database and you need to create additional containers to run additional services without having to start each one separately. you would write a set of files using Docker compose to do that.

Docker Swarm

Docker swarm is a service for containers that allows IT administrators and developers to create and manage a cluster of swarm nodes within the Docker platform. Each node of Docker swarm is a Docker daemon, and all Docker daemons interact using the Docker API. A swarm consists of two types of nodes: a manager node and a worker node. A manager node maintains cluster management tasks. Worker nodes receive and execute tasks from the manager node.

After having looked into all the components of Docker, let us advance our learning in this getting started with docker tutorial on the Docker commands and use case.

Docker Desktop

It’s easy to get overwhelmed by terminology, and some people may wonder if Docker (also known as Docker Engine) and Docker Desktop are different things. However, the two entities aren’t the same. Docker Desktop is a free, easy-to-install, downstream application for a Mac or Windows environment. The application lets you build and share containerized applications and microservices. Docker consists of[Docker Engine](https://docs.docker.com/engine/),[Docker Compose](https://docs.docker.com/compose/), Docker CLI client, [Docker Content Trust](https://docs.docker.com/engine/security/trust/), [Kubernetes](https://github.com/kubernetes/kubernetes/), and [Credential Helper](https://github.com/docker/docker-credential-helpers/).

Docker Desktop functions well with any developer’s choice of tools and languages and gives access to a massive library of certified images and templates residing in [Docker Hub](https://hub.docker.com/). These resources help development teams extend their environment to rapidly auto-build, continuously integrate, and collaborate with a secure repository.

So, to recap, Docker Engine is the client-server technology used to build and run containers and is often referred to as “Docker.” Additionally, people use “Docker” to refer to the company itself. On the other hand, Docker Desktop is a free application that bundles together numerous Docker resources. Phew! Alright, let’s move on.

Orientation and Setup

Naturally, when getting started with Docker, the first step is installing it if you haven’t already. Here’s how to install Docker on[Windows](https://www.simplilearn.com/tutorials/docker-tutorial/install-docker-on-windows) and[Ubuntu](https://www.simplilearn.com/tutorials/docker-tutorial/how-to-install-docker-on-ubuntu) (Linux OS). Check[here](https://docs.docker.com/desktop/mac/install/) if you’re using a Mac.

Here is the run command you need to get the tutorial started:

docker run -d -p 80:80 docker/getting-started

Additionally, you should ensure that the Docker dashboard is running. To access the Docker dashboard, follow the directions shown in the [Docker Desktop manual](https://docs.docker.com/desktop/dashboard/), available on the Docker site. We will talk more about the Docker dashboard later.

Make sure you have a supported version of Docker by running this command:

docker –- version

You can find out more details about your installed version by running:

docker info

Alternatively, you can do it by running the Docker –-version command but without the “- -“  dashes.

Docker Dashboard

The Docker dashboard gives you a quick view of any containers running on your machine. The Docker Dashboard is available for Windows and Mac. The dashboard lets you get a shell inside the container, gives you rapid access to container logs, and allows you to manage your containers' lifecycles easily.

Once you have installed Docker, you can test it by running this basic Docker image:

docker run hello-world

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

ca4f61b1923c: Pull complete

Digest: sha256:ca0eeb6fb05351dfc8759c20733c91def84cb8007aa89a5bf606bc8b315b9fc7

Status: Downloaded newer image for hello-world:latest

Hello from Docker!

This message shows that your installation appears to be working correctly.

...

If Docker is working correctly, list the hello-world image downloaded to your system.

docker image ls

Finally, list the hello-world container generated by the image which exits after displaying the message.

 docker container ls --all

CONTAINER ID IMAGE       COMMAND  CREATED        STATUS

54f4984ed6a8 hello-world "/hello" 20 seconds ago Exited (0) 19 seconds ago

All text boxes courtesy of[Docker.com](https://docker-docs.netlify.app/get-started/#recap-and-cheat-sheet).

And congratulations! You have set up your Docker environment!

Choose a Plan That is Right for You

Docker offers different pricing plans to fit anyone's needs. Before you make a choice, take some time and evaluate your needs. How many development projects do you and your team do regularly? The trick is to ensure you don't overspend or subscribe to a plan that includes features you'd never use.

Here’s a subscription breakdown.

* Personal: Cost: Free. This plan is excellent for individual developers, educators, open-source communities, and small businesses. It includes Docker Desktop.
* Pro: Cost: $5/month. This plan is tailored for individual developers and includes Docker Desktop and other tools to accelerate productivity.
* Team: Cost: $7 per user per month. This plan is perfect for small development teams and workgroups of five or more members. It also includes Docker Desktop and increased security, productivity, and collaboration.
* Business: Cost: $21 per user per month. This plan is geared towards medium and large companies for teams of 50 or more. Like the other plans, it includes Docker Desktop and features centralized management and enhanced security measures.

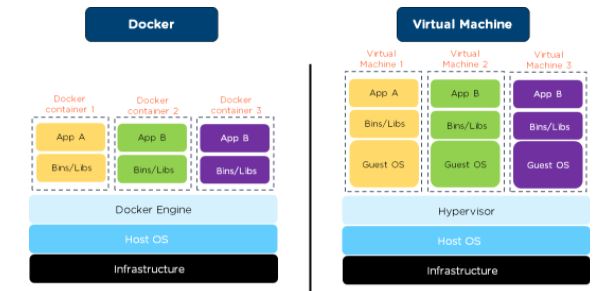
What is a Virtual Machine?

A virtual machine (VM) is a computing environment or software that aids developers to access an operating system via a physical machine.

Now, let’s dig into the concept of Docker vs. virtual machine.

Docker vs. Virtual Machine

Depicted below is a diagrammatic representation of how an application looks when deployed on Docker and virtual machines:



Now, let's have a look at the primary differences between Docker and virtual machines.

|  |  |  |
| --- | --- | --- |
| Differences | Docker | Virtual Machine |
| Operating system | Docker is a container-based model where containers are software packages used for executing an application on any operating system  In Docker, the containers share the host OS kernel  Here, multiple workloads can run on a single OS | It is not a container-based model; they use user space along with the kernel space of an OS  It does not share the host kernel  Each workload needs a complete OS or hypervisor |
| Performance | Docker containers result in high-performance as they use the same operating system with no additional software (like hypervisor)  Docker containers can start up quickly and result in less boot-up time | Since VM uses a separate OS; it causes more resources to be used  Virtual machines don’t start quickly and lead to poor performance |
| Portability | With docker containers, users can create an application and store it into a container image. Then, he/she can run it across any host environment  Docker container is smaller than VMs, because of which the process of transferring files on the host’s filesystem is easier | It has known portability issues. VMs don’t have a central hub and it requires more memory space to store data  While transferring files, VMs should have a copy of the OS and its dependencies because of which image size is increased and becomes a tedious process to share data |
| Speed | The application in Docker containers starts with no delay since the OS is already up and running  These containers were basically designed to save time in the deployment process of an application | It takes a much longer time than it takes for a container to run applications  To deploy a single application, Virtual Machines need to start the entire OS, which would cause a full boot process |

Key Difference: Docker and Virtual Machine

There are many analogies of Docker and virtual machines. Docker containers and virtual machines differ in many ways; let's discuss one analogy using apartment vs. bungalow.

|  |  |
| --- | --- |
| Apartment (Eg: Containers) | Virtual machine (Eg: Bungalow) |
| Most amenities (binary and library) are shared with neighbors (applications) | Amenities (binary and library) cannot be shared with neighbors (applications) |
| Can have multiple tenants (Applications) | Cannot have multiple tenants (application) |

For a more in-depth understanding, we will look at the key differences between the two below:

|  |  |
| --- | --- |
| Docker | Virtual machine |
| Containers stop working when the “stop command” is executed | Virtual machines are always in the running state |
| It has lots of snapshots as it builds images upon the layers | Doesn’t comprise many snapshots |
| Images can be version controlled; they have a local registry called Docker hub | VM doesn’t have a central hub; they are not version controlled |
| It can run multiple containers on a system | It can run only a limited number of VMs on a system |
| It can start multiple containers at a time on the Docker engine | It can start only a single VM on a VMX |

Next, let’s have a look at a real-life use-case of Docker using the BBC news channel.

Real-Life Use-Case of Docker

BBC is a British news channel with over 500 developers working across the globe. Let’s move forward and understand what issues the company had and how Docker resolved them.

Challenges

* BBC uses several languages in different areas of the world because of which it consists of over 10 Continuous Integrations
* The company had to identify a way to unify the coding processes and monitor the Continuous Integration consistently
* Also, the existing jobs took up to 60 minutes to schedule and perform its task

Solution

* With the help of containers, the developers were able to work in flexible CI environments
* Entire code processes were unified and stored in a single place for easy and quick access
* Docker helped in eliminating job wait time and resulted in speeding up the entire process.