**DOCKER**

**What is Docker** : Docker is an open source, Docker is a platform for creating virtual environments which are extremely lightweight virtual machines. Docker allows developers and system administrators to quickly assemble, test, and deploy applications and their dependencies inside Linux containers supporting the multi-tenancy deployment model on a single host. Docker’s lightweight containers lend themselves to rapid scaling up and down. A container is a group of controlled processes associated with a separate tenant executed in isolation from other tenants. It is written in the Go programming language.

Docker runs with any modern-kernel 64-bit Linux operating system.

Docker runs natively on Linux, it may be desirable to have two different host environment, one is Ubuntu and second is CentOS. To achieve this, VMs running Docker may be used. To simplify management of different Docker host, it is possible to use Docker Machine. Docker Machine is a tool that lets you install Docker Engine on virtual hosts, and manage the hosts with Docker machine commands. System virtualization tools or emulators such as virtual packer, Hyper-V or V M ware, boot virtual machines from a complete guest OS image and emulate a complete machine, which results in a high operational overhead. Virtual environments created by Docker run on the existing operating system kernel of the host’s OS without a need for a hypervisor. This leads to very low overhead and significantly faster container start up time. Docker-provisioned containers do not include or require a separate operating system. This circumstance puts a significant limitation on your Operating systems.

**Why use Docker and pros :**

This tool can really change a developer’s daily life. In order to best answer this question, I have written a non-exhaustive list of the benefits you will find:

1. Docker is fast. Unlike a virtual machine, your application can start in a few seconds and stop just as quickly.

2. Docker is multi-platform. You can launch your container on any system.

3. Containers can be built and destroyed faster than a virtual machine.

No more difficulties setting up your working environment. Once your Docker is configured, you will never have to reinstall your dependencies manually again. If you change computers or if an employee joins your company, you only have to give them your configuration.

4. You keep your work-space clean, as each of your environments will be isolated and you can delete them at any time without impacting the rest.

5. It will be easier to deploy your project on your server in order to put it online.

**Container and Image:**

A Docker container is a virtualized runtime environment used in application development. It is used to create, run and deploy applications that are isolated from the underlying hardware. A Docker container can use one machine, share its [kernel](https://www.techtarget.com/searchdatacenter/definition/kernel) and virtualize the OS to run more isolated processes. As a result, Docker containers are light weight.

A Docker image is like a snapshot in other types of VM environments. It is a record of a Docker container at a specific point in time. Docker images are also immutable. While they can't be changed, they can be duplicated, shared or deleted. The feature is useful for testing new software or configurations because whatever happens, the image remains unchanged.

Containers need a runnable image to exist. Containers are dependent on images, because they are used to construct runtime environments and are needed to run an application.

**Docker Image :**

A [Docker](https://www.techtarget.com/searchitoperations/definition/Docker) image is a file used to execute code in a Docker container. Docker images act as a set of instructions to build a Docker [container](https://www.techtarget.com/searchitoperations/definition/container-containerization-or-container-based-virtualization), like a template. Docker images also act as the starting point when using Docker. An image is comparable to a snapshot in virtual enviroment machines.

Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container.

Docker images have multiple layers, each one originates from the previous layer but is different from it. The layers [speed up Docker builds](https://www.techtarget.com/searchitoperations/tip/Optimize-Docker-images-for-improved-efficiency-and-security) while increasing reusability and decreasing disk use. Image layers are also read-only files. Once a container is created, a writable layer is added on top of the unchangeable images, allowing a user to make changes.

References to disk space in Docker images and containers can be confusing. It's important to distinguish between size and virtual size. Size refers to the disk space that the writable layer of a container uses, while virtual size is the disk space used for the container and the writeable layer. The read-only layers of an image can be shared between any container started from the same image.

**Image Use Cases :**

A Docker image has everything needed to run a containerized application, including code, [configration files](https://www.techtarget.com/searchitoperations/definition/configuration-file), environment variables, libraries and runtimes. When the image is deployed to a Docker environment, it can be executed as a Docker container. The [Docker run command](https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/Docker-run-vs-docker-compose-Whats-the-difference) creates a container from a specific image.

Docker images are a reusable asset -- deployable on any host. Developers can take the static image layers from one project and use them in another. This saves the user time, because they do not have to recreate an image from scratch.

**Docker Container :**

Docker Container is a standardized unit which can be created on the fly to deploy a particular application or environment. It could be an Ubuntu container, Cent Os container, etc. to full-fill the requirement from an operating system point of view. Also, it could be an application oriented container like Cake PHP container or a Tomcat-Ubuntu container etc.

**System Requirements For Docker Installation:**

1. 64-bit Windows 10 and the Hyper-V and Containers Windows features must be enabled.

2. As far as RAM is concerned a minimum of 4 GB RAM is required. Do not worry about enabling the Windows Hyper-V feature, the Docker desktop application will take care of this during installation.

Installation:

Installation of Docker varies greatly depending on the operating system your are using. But it's universally simple across the board. Docker runs flawlessly on all three major platforms, Mac, Windows, and Linux.

**Installation (Windows):**

On Windows, the procedure is almost the same, except there are a few extra steps that you’ll need to go through.

Step 1: Double-click Docker Desktop Installer.exe to run the installer.

Step 2: When prompted, ensure the Use WSL 2 instead of Hyper-V option on the Configuration page is selected or not depending on your choice of back end.

Step 3: Follow the instructions on the installation wizard to authorize the installer and proceed with the install.

Step 4: When the installation is successful, click Close to complete the installation process.

Step 5: If your admin account is different to your user account, you must add the user to the Docker users group. Run Computer Management as

an admin and navigate to local users and groups>groups> Docker-users Right-click to add the user to the group. Log out and log back in for the changes to take effect.

**Commands and Installation of Docker :**

Through the command prompt in a terminal to install Docker Desktop version

Command: "Docker Desktop Installer.exe" install

Command: Start-Process 'Docker Desktop Installer.exe' -Wait install

Command: start /w "" "Docker Desktop Installer.exe" install

**Docker Desktop icon:**

Docker Desktop does not start automatically after installation. To start Docker Desktop

**Edit and Creating Docker Images:**

A Docker image is a read-only template that contains a set of instructions for creating a container that can run on the Docker platform. It provides a convenient way to package up applications and preconfigured server environments, which you can use for your own private use or share publicly with other Docker users. Docker images are also the starting point for anyone using Docker for the first time.

So, in this introduction, we’ll not only take you through the basics of Docker images, but also show you where to find ready-made, off-the-shelf images that will give you a head start in building your own containerized applications, tools, and services.

As a new Docker user, you’ll also need to understand how to build your own custom images. So, we’ll briefly cover how to create Docker images for deploying your code and assembling container-based services. But first let’s look at the composition of a Docker image in more detail.

**blinding and shipping image docker :**

A Docker image is made up of a collection of files that bundle together all the essentials – such as installations, application code, and dependencies required to configure a fully operational container environment. You can create a Docker image by using one of two methods

Interactive method: By running a container from an existing Docker image, manually changing that container environment through a series of live steps, and saving the resulting state as a new image.

**Docker file method :**

By constructing a plain-text file, known as a Dockerfile, which provides the specifications for creating a Docker image.

**Layers (Image) :**

Each of the files that make up a Docker image is known as a layer. These layers form a series of intermediate images, built one on top of the other in stages, where each layer is dependent on the layer immediately below it. The hierarchy of your layers is key to efficient lifecycle management of your Docker images. Thus, you should organize layers that change most often as high up the stack as possible. This is because, when you make changes to a layer in your image, Docker not only rebuilds that particular layer, but all layers built from it. Therefore, a change to a layer at the top of a stack involves the least amount of computational work to rebuild the entire image.

**Layers (Container) :**

Each time Docker launches a container from an image, it adds a thin writable layer, known as the container layer, which stores all changes to the container throughout its runtime. As this layer is the only difference between a live operational container and the source Docker image itself, any number of like-for-like containers can potentially share access to the same underlying image while maintaining their own individual state.

**Create Project on Docker :**

In order to create your first Docker application, I invite you to create a folder on your computer. It must contain the following two files:

1. ‘main.py’ file is python file that will contain the code to be executed.

2. Docker file that will contain the necessary instructions to create the environment.

**Create the Docker image :**

Once your code is ready and the Dockerfile is written, all you have to do is create your image to contain your application.

>> docker build -t python-test .

The ’–t ’option allows you to define the name of your image. In our case we have chosen ’ python-test’ but you can put what you want.

**Run the Docker image :**

Once the image is created, your code is ready to be launched.

>>docker run python-test

You need to put the name of your image after docker run.

There you go, that’s it. You should normally see “Docker is magic!” displayed in your terminal.

If you want to retrieve the complete code to discover it easily or to execute it, I have put it at your disposal on my GitHub.

[**GitHub: Docker First Application example**](https://github.com/gael-thomas/Docker-First-Application-example) **:**

Before I leave you, I have prepared a list of commands that may be useful to you on Docker.

1. List your images and its command.

>> docker image ls

1. Delete a specific image.

>> docker image rm [image name]

1. Delete all existing images.

>> docker image rm >> (docker images -a -q)

1. List all existing containers (running and not running).

>> docker ps -a

1. Stop a specific container.

>> docker stop [container name]

1. Stop all running containers.

>> docker stop >> (docker ps -a -q)

1. Delete a specific container (only if stopped).

>> docker rm [container name]

1. Delete all containers (only if stopped).

>> docker rm >> (docker ps -a -q)

1. Display logs of a container.

>> docker logs [container name]

Next>>

After all your feedback, I decided to write the next part of this beginner’s guide. In this article, you will discover how to use docker - compose to create your first client/server-side application with Docker.

**Asyncio**

The Asyncio module was added to Python in version 3.4 as a provisional package. What that means is that it is possible that Asyncio receives backwards incompatible changes or could even be removed in a future release of Python.

**Definition Of Asyncio :**

The Asyncio module provides a framework that revolves around the event loop. An event loop basically waits for something to happen and then acts on the event. It is responsible for handling such things as I/O and system events.

**Async and Await :**

The async and await keywords were added in python 3.5 to define a native coroutine and make them a distinct type when compared with a generator based coroutine.If you did like an in depth description of async and await

The async and await keywords can be considered an API to be used for asynchronous programming. The asyncio module is just a framework that happens to use async and await or programming asynchronously. There is actually a project called curio that proves this concept as it is a separate implementation of an event loop thats uses async and await under neath the covers.

**Python async functionality:**

1. We have imported the asyncio module to get access to Python async functionality.

2. Then create a primary() function and write the async keyword in front of that. This will allow the program to run the task asynchronously.

3. We used for loop and called the sleep() method, which forced us to wait 1 second.

4. The program prints "Hello" after one second.

5. Program should have the one .run() function as well as one **.**main() function.

Example :

Import asyncio

async def main():

Print(“Ravi”)

await asyncio.sleep(3)

print(“Chandra”)

asyncio.run(main())

output >> Ravi

#after 3 sec

Chandra

“””------“””

**Parallelism:**

**Parallelism is** involves performing multiple operations at a time. Multiprocessing is an example of it. It is well suited for CPU bound tasks.

**Concurrency :**

**Concurrency**is slightly broader than Parallelism. It involves multiple tasks running in an overlapping manner.

**Threading:**

A thread is a separate flow of execution. One process can contain multiple threads and each thread runs independently. It is ideal for IO bound tasks.

**components of Async:**

Let’s explore the various components of Async IO in depth. We will also look at an example code to help us understand the implementation.

**Coroutines:**

Coroutines are mainly generalization forms of subroutines. They are generally used for cooperative tasks and behave like Python generators.

An async function uses the await keyword to denote a coroutine. When using the awaitkeyword, coroutines release the flow of control back to the event loop.

To run a coroutine, we need to schedule it on the event loop. After scheduling, coroutines are wrapped in Tasks as a Future object.

Example:

Import asyncio

Async def async\_func():

Print(‘python’)

await asyncio.sleep(3)

print(“version”)

async def main():

async\_func()

await async\_func()

asyncio.run(main())

output>> C:

C:\Users\HP\PycharmProjects\sample\main.py:980: RuntimeWarning: coroutine 'async\_func' was never awaited

async\_func()#this will do nothing because coroutine object is created but not awaited

RuntimeWarning: Enable tracemalloc to get the object allocation traceback

Velotio ...

... Technologies!

Process finished with exit code 0

Tasks:

Tasks are used to schedule coroutines concurrently.

When submitting a coroutine to an event loop for processing, you can get a Task object, which provides a way to control the coroutine’s behavior from outside the event loop.

Example:

import asyncio

async def async\_func():

print('cricket')

await asyncio.sleep(3)

print('wicket')

async def main():

task = asyncio.create\_task (async\_func())

await task

asyncio.run(main())

output>> cricket

(after 3 sec )

wicket

Process finished with exit code 0

Loop:

This mechanism runs coroutines until they complete. You can imagine it as while(True) loop that monitors coroutine, taking feedback on what’s idle, and looking around for things that can be executed in the meantime.

Manage an async event loop in Python :

Asyncio is also used for managing the async event loop. An event loop is an object that runs async functions and call backs. When we want to execute the coroutines, the event will be crucial for the asynchronous functions when we run the **asyncio.run()** method, the event loop object is created automatically. To implement the more advanced server, we will need lower-level access to the event loop. We need to work directly with the event loop's internals.

Event loop come :

1. It can register, execute, and cancel delayed calls asynchronous functions.

2. It can create client and server transports for communication.

3. It can create sub processes and transports for communication with another program.

4. Delegate function calls to a pool of threads.

Synchronization tasks in Python :

We have discussed earlier, Asynchronous program runs separately, but sometimes we would want to communicate with each other. The asyncio module offers us a queue and various other methods to establish the synchronization between the tasks.

**1. Queues -** The asyncio queues facilitate asynchronous functions to line up Python objects to be consumed by other async functions. For example - The workload is distributed between the function on its behavior.

**2. Synchronization Primitive -** The asyncio's features locks, events, conditions, and semaphores act as conventional Python counterparts.

Here, a point should always keep in mind that these methods are not **thread-safe.** This isn't an issue for async tasks running in the same event loop. But we need to use the thread module to share the information between the tasks.

Use asynchronous in program:

In the following scenario.

(A). When we want complete work in quick time.

(B). The delay involves waiting for I/O (disk or network) operations, not computation.

(C). When many I/O operations are happing at once.

The Asyncio module :

1. Web scraping.

2. Network Services (web server and framework)

3. Simultaneous Database

Conclusion:

This tutorial includes the concept of asynchronous programming using the Python asyncio module. The asyncio gives us programmatic control of when context when we use the context switches. That means we can handle many complex issues that occur with threaded programming.

It is a powerful and valuable tool, but only for asynchronous type programming. We have discussed coroutines and tasks with their respective example. We have also discussed managing the event loop and reading and writing data with stream in Python. It also includes the essential methods.