Learn about Docker

By

Stanley Stephen

Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca/Learn_from_Stanley/
Email: s.stanley.mca@gmail.com

Overview about Docker

What is Docker?

Docker is a computer program that performs operating-system-level virtualization, also known as “containerization”.

More than that, Docker is a popular tool to make it easier to build, deploy and run applications using containers. Containers allow us to package all the things that our application needs like such as libraries and other dependencies and ship it all as a single package. In this way, our application can be run on any machine and have the same behaviour.

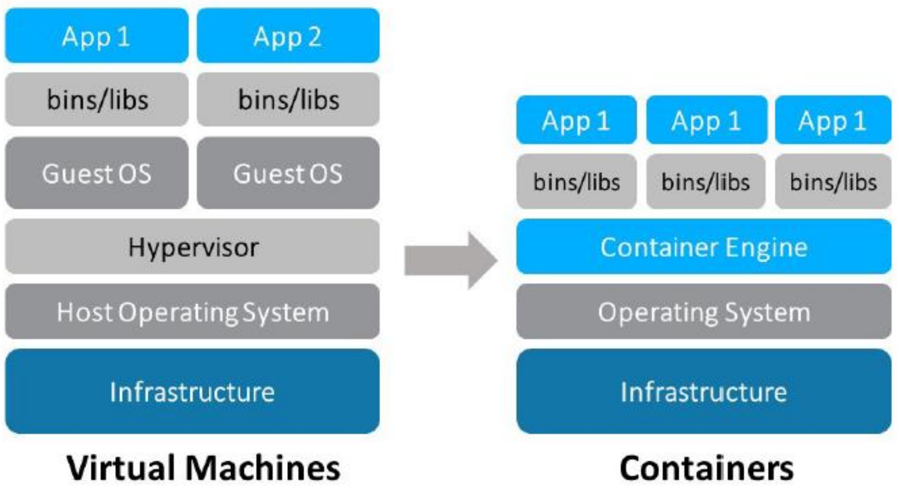
Docker was first released in 2013, only 5 years ago, but it is already used by a lot of companies, like: Spotify, The New York Times, PayPal, Uber and more (https://www.contino.io/insights/whos-using-docker).

Last but not least, Docker is open source: https://github.com/docker.

Please note that Docker is not a virtual machine (VM).

A Docker container, unlike a virtual machine, does not require or include a separate operating system. Instead, it relies on the kernel’s functionality and uses resource isolation for CPU and memory, and separate namespaces to isolate the application’s view of the operating system.

Based on the short description above, the following image shows a comparison between virtual machines and Docker containers.



Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca/Learn_from_Stanley/Docker_by_Stanley.pdf
Email: s.stanley.mca@gmail.com

As we can see, Docker containers are simpler than virtual machines and using it we can avoid the overhead of starting and maintaining VMs.

As this is a hands-on tutorial, I will not go deep on how Docker works under the hood. If you want to learn more about it I suggest you read the Docker documentation.

Hands-on

Now that we have learned what is Docker, let's start the hands-on tutorial.

Installation

Since the installation depends on your operating system, we will not cover it on this tutorial. To install Docker on your OS please follow the official docs:

- Install Docker for Mac
- Install Docker for Windows
- Get Docker CE for Ubuntu

There is also a great tutorial in How to Install and Use Docker on Ubuntu 16.04 (Digital Ocean).

Hello World

With Docker properly installed and running, let's start creating containers.

The "Hello World" in Docker is simple like that:

```
$ docker run hello-world
```

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
9db2ca6ccae0: Pull complete
Digest:
sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afa
cdc
Status: Downloaded newer image for hello-world:latest
```

```
Hello from Docker!
This message shows that your installation appears to be working
correctly.
...
```

As you can see, it shows the following message:
Unable to find image 'hello-world:latest' locally

It means you haven't an image called "hello-world" locally so it will automatically pull from Docker hub.

Docker hub is basically:

A cloud-based registry service which allows you to link to code repositories, build your images and test them, stores manually pushed images, and links to Docker Cloud so you can deploy images to your hosts. It provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline.

After pulling the *hello-world* image, it will run the container and will show a *Hello from Docker!* message with some other information.

Congratulations, you have run a "Hello World" in Docker!

Creating and Running Containers

Now that we have already created our "Hello World" example and we know basically what is Docker hub, let's create something a little more sophisticated. Let's create a simple Flask application in Python.

At this point, it would be great to have a little experience with Python and Flask to know what is going on here, but don't worry about it, we just need to know that it is a "Hello World" web application in Flask which will run on localhost from inside a container.

First of all, we need to create a directory called *my_web_app* and save the following code in a file called *app.py* .

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
	return "Hello World!"

if __name__ == "__main__":
	app.run(debug=True, host="0.0.0.0")
```

The Python code is just that. It will import the Flask package, create a Flask *app* , define a route and run the application in debug mode on localhost.

So, let's start creating our Dockerfile. To do that, just create a file named *Dockerfile* in the same directory of the Python file, and put the following commands inside it:

```
# Inherit from the Python Docker image
FROM python:3.7-slim
# Install the Flask package via pip
RUN pip install flask==1.0.2
# Copy the source code to app folder
COPY ./app.py /app/
# Change the working directory
WORKDIR /app/
# Set "python" as the entry point
ENTRYPOINT ["python"]
# Set the command as the script name
CMD ["app.py"]
```

As we can see by the comments, it will inherit from the Python Docker image, install the Flask package, copy the source code of our application to the `app` folder, change the working directory to the `app` folder and set an *ENTRYPOINT* and a *CMD*.

Now let's build the image from the `Dockerfile` using the `-t` flag, which means *tag,* and set a name (*flask_app*) and a tag (*0.1):*

```
$ docker build -t flask_app:0.1 .
$ docker images
```

```
REPOSITORY    TAG         IMAGE ID        CREATED          SIZE
flask_app     0.1         c6eb89fefb25    2 minutes ago    153MB
python        3.7-slim    4c2534c95211    4 weeks ago      143MB
```

By typing *docker images* we can see that the image was successfully created. Then let’s run the container specifying the port that will be mapped *(-p)* and using the -d flag, which means *detached*, so that the terminal does not get stuck. We must also pass the name and tag of the image as parameter (*flask_app:0.1*).

```
$ docker run -d -p 5000:5000 flask_app:0.1
$ docker ps

CONTAINER ID    IMAGE            COMMAND           CREATED
03c650a4eb58    flask_app:0.1    "python app.py"   16 seconds ago

STATUS          PORTS                       NAMES
Up 3 seconds    0.0.0.0:5000->5000/tcp      gifted_kar
```

The *docker ps* command will show the running container with some other information like the container ID, the image that the container is using, the command the container is executing, the time it was created, the current status, the ports that were mapped and the name (in this case a random name) of the container (we can also set a name for the container by using the --*name* flag).

Open your browser, go to the *localhost:5000* address and voilà, we are accessing our web app which is running inside the container.

Stopping and Removing Containers

As our container is still running, we can stop it before deleting. To do so, we just need to run *docker* stop passing the container name as parameter, for example:

```
$ docker stop gifted_kar
```

We can also stop the container by using the container ID, for example:

```
$ docker stop 03c650a4eb58
```

If we run *docker ps -a* right now we will see that the container is in that list with status *Exited*. Let's delete the container and then delete the image.

Similar to the *docker stop* command, we can use the *docker rm* command passing the container name or ID as the parameter, for example:

```
$ docker rm gifted_kar
# or
$ docker rm 03c650a4eb58
```

If we run *docker ps -a* now we will see that our container has disappeared from the list.

If we run the *docker images* command we will see our *flask_app* with tag 0.1 in that list. To remove our *flask_app* image we can use the *docker rmi* command, for example:

```
$ docker rmi flask_app:0.1
```

If we run *docker images* again we will see that our *flask_app* image was removed. As we have seen, the Docker command line interface is very simple to use. You can continue exploring Docker commands by following the Docker documentation.

Basic Commands

In this section you may find some basic Docker commands, like build, run and exec, amongst others.

```
# Build a Docker image
$ docker build -t [image_name]:[tag] .
# Run a Docker container specifying a name
$ docker run --name [container_name] [image_name]:[tag]
# Fetch the logs of a container
$ docker logs -f [container_id_or_name]
# Run a command in a running container
$ docker exec -it [container_id_or_name] bash
# Show running containers
$ docker ps
# Show all containers
$ docker ps -a
# Show Docker images
```

```
$ docker images
# Stop a Docker container
$ docker stop [container_id_or_name]
# Remove a Docker container
$ docker rm [container_id_or_name]
# Remove a Docker image
$ docker rmi [image_id_or_name]
```

Every command has it helps page, so you can call for example:

```
$ docker build --help
$ docker run --help
```

And see the OPTIONS you can pass to it.

Useful Commands

Some other useful commands to perform operations in multiple items:

```
# Stop all running containers
$ docker stop $(docker ps -q)

# Remove all containers
$ docker rm $(docker ps -aq)

# Remove all images
$ docker rmi $(docker images -aq)
```

Integration

Docker can be integrated into various infrastructure tools, like Google Cloud Platform (GCP), Amazon Web Services (AWS), Microsoft Azure, Oracle Container Cloud Service and others.

We can easily use Docker to apply Continuous Integration (CI) and Continuous Deployment (CD) practices, making our life easier and the deployment process faster. If you’re not familiar with CI and CD, please refer my post on CICD that can help you understand the difference between both.

Stanley Stephen, M.C.A.,
Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca/Learn_from_Stanley/Docker_by_Stanley.pdf
Email: s.stanley.mca@gmail.com