THE UNIVERSITY OF SYDNEY

School of Information Technologies
Dr. Ying Zhou

COMP5349: Cloud Computing **Sem. 1/2018**

Week 3: Docker Tutorial

22.03.2018

In this lab, you will learn the basic usage of Docker. You are expected to get familiar with its key component including docker daemon, registry, containers and images by going through the official tutorial. You will also gain some experience with the Docker file system and storage drivers using standard and customised Jupyter notebook container. The namespace feature of Linux kernel will be briefly explored as well.

Please note that for security reasons, Docker cannot run on lab workstations directly. Instead, you will use Docker inside a virtual machine.

Question 1: Starting up a Virtual Machine

We have `VirtualBox 5.0` running on the RHEL workstations in all labs. A Ubuntu 16.0.4 LTS image has been set up for VirtualBox. The image is located on a network drive `/labcommon/comp5349`. You need to copy it to your local workstation using the following command:

```
cp -r /labcommon/comp5349/docker/ubuntudocker /tmp/ubuntudocker.$USER
```

This may take a couple of minutes as the image file is quite large. Once finished, the image would be stored in `/tmp/ubuntudocker.<your_unikey>` directory. Run `VirtualBox` to start the VirtualBox Hypervisor. Click OK to the "upgrade available" message if there is any, then under "Machine" click "Add" to add the VM image copy by navigating to `/tmp/ubuntudocker.<your_unikey>` and selecting the `ubuntudocker.vbox` file.

It will appear in the set of VMs in VirtualBox.

Select Ubuntu Guest OS and click "Start". This may take a while then you will see the `Ubuntu` desktop. The Docker daemon is started when the VM is booted. You are logged in as `student`, the password is "student". The `student` user is added to `sudo` group.

In the following exercises, you will need to use the `Terminal` application inside the VM to run most commands and a browser inside it to visit various container applications.

All `docker` command requires a `sudo` privilege. Hence either run `sudo -i` just after you start Terminal or remember to prepend `sudo` to all `docker` commands. The sudo password is "student".

At the end of the lab please remember to **clean up** by firstly, shutting down the Ubuntu OS, powering off the VM, then in VirtualBox select the image again, and right click to remove it, selecting the option to remove all files. We ask you to remove the image so that the `/tmp` directory do not fill up quickly.

Question 2: Basic Docker Usage

Please follow the official docker tutorial to get familiar with basic docker usage: `https://docs.docker.com/get-started/part2/`.

Optional: Copy pasting between your host OS and the Guest OS is not setup by default. Either setup a shared clipboard by following the instruction here: `https://www.liberiangeek.net/2013/09/copy-paste-virtualbox-host-guest-machines/`, or just open the tutorial on a browser installed in the Guest OS itself.

Question 3: Using Volumes in Docker

This exercise demonstrates the usage of Volumes (`https://docs.docker.com/storage/volumes/`) through a Jupyter Notebook container. You may build your own image from scratch. However, most of the time we just prepare an image from some base images published by some trusted parties. Typical base image would be a basic OS (e.g. Ubuntu) or some language runtime like `python 3.6`. It is possible to use other, more complicated image as a base. The procedure to set up container image is similar to procedure of installing software to set up a working environment.

To find suitable base images, you can do a general Google search, or go directly to the organisation managing the software to see if they have a Docker version published. A more efficient way is to use the

`docker search` command, which searches the Docker Hub for images.

a) **Starting a Jupyter Notebook Container**
Use the following `docker search` command to find possible jupyter related images:

```
docker search jupyter
```

This would give you a long list of images whose name contains 'jupyter'. We only need the very basic one. There are two possible options: `jupyter/minimal-notebook` and `jupyter/base-notebook`. We can find detailed information from `https://github.com/jupyter/docker-stacks`. It turns out the smallest option is `jupyter/base-notebook` and it is built on top of a `ubuntu` image. We will use that as our image.

Run the following command to pull the image into the local registry

```
docker pull jupyter/base-notebook
```

After finishing, you can inspect its size using `docker images` command, which should show you something similar to

```
jupyter/base-notebook   latest   f5e42ba6962e   11 hours ago    697MB
```

It is not a very small image. To inspect each layer and the commands used to build those layers, use:

```
docker history jupyter/base-notebook
```

You will see a list of command each add a layer to the image. The largest two representing the base OS and the miniConda installation. Alternatively, you can view the actual `Dockerfile` from the container's github repository.

An image publisher usually provides a `README` file. It usually contains a brief description and usage details as well as command examples. For instance, the basic command to start this container is as follow:

```
docker run -it --rm -p 8888:8888 jupyter/base-notebook
```

As described in the `README` file, "the command starts a container with the Notebook server listening for HTTP connections on port 8888 with a randomly generated authentication token configured." Once the container is successfully started, you will see a message like:

```
Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
    http://localhost:8888/?token=xxxxx
```

Just do as instructed. The note book is running from local directory `/home/jovyan`. A sub folder `work` has been created under this home directory. When you open the notebook from a browser, you will see this folder.

Create a new note book inside this folder. You can write any simple program as you like. Below is an example for producing a list of packages installed. you can give it a name 'get_packages'

Type the following in the first code cell and run it:

```
import pip
installed_packages = pip.get_installed_distributions()
for i in installed_packages:
    installed_packages_list = sorted(["%s==%s" % (i.key, i.version)])
    print(installed_packages_list)
```

Save the notebook and go back to the home screen, you will see a new file `get_packages.ipynb` under `work`. This file is stored currently at the thin writable layer of this container. After you exit the container, the thin writable layer will be deleted, and hence the Jupyter note book we created will disappear.

Exit the container by pressing `CTR+C` on the terminal you started the container. Use the `docker run` command to restart the container, you will find the `work` directory is empty again.

b) **Creating a Volume and Attach it to a Container**

There are various ways to instruct a container to use host file system as storage. The container's `README` provides the following option:

```
-v /some/host/folder/for/work:/home/jovyan/work
```

This is called bind mount, which directly mounts a host machine directory as folder in the container. However, using Volume is a preferred way.

Below is the instruction to create a volume called `jupyter-work`.

```
docker volume create jupyter-work
```

The volume resides in a designated location in the host machine. The location is configurable. After the volume is created, we can attach it to the container when starting it. In the following command we add an `-v` option when starting the notebook:

```
docker run -it --rm -p 8888:8888 -v jupyter-work:/home/jovyan/work
jupyter/base-notebook
```

The `-v jupyter-work:/home/jovyan` option specifies that the container's `jupyter-work:/home/jovyan/work` directory should be populated with volume `jupyter-work`

To check if the volume is successfully mounted, run the following command to find out your container's ID.

```
docker container ls
```

You can find the ID value from the first column. The `docker inspect` command would print out information about the container specified by its ID:

```
docker inspect <container_id>
```

Look for the Mounts section, you are likely to see something like:

```
"Mounts": [
            {
                "Type": "volume",
                "Name": "jupyter-work",
                "Source": "/var/lib/docker/volumes/jupyter-work/_data",
                "Destination": "/home/jovyan/work",
                "Driver": "local",
                "Mode": "z",
                "RW": true,
                "Propagation": ""
            }
        ],
```

Now repeat what you have done in the previous exercise to create and save a very simple notebook under the `work` directory. After you've done, exit and restart the container with the same volume attached, you will see the notebook there.

Question 4: Create Your Own Notebook Container

To create a customised notebook container with some sample notebooks, the actual notebook files need to be written in the image. In this exercise, we will create a container with

the notebook in week1 python exercise. We can use the `jupyter/base-notebook` as the parent image.

If you have not done so, clone the python resources repository to a local directory:

```
git clone
https://github.sydney.edu.au/COMP5349-Cloud-Computing/python_resources.git
```

You will be at the master branch by default, now check out the `solutions` branch. This branch contains solutions to the exercise.

Create a directory to store all files needed for your new image. Let's call it `jupyter_solutions`. Copy `week1` from `python_resources` directory into `jupyter_solutions` directory. These are files we want to write into the new image.

Building this image involves only two steps:

1. Grab a base Jupyter notebook container
2. Copy the files to the working directory

These are translated into two commands in `Dockerfile`:

```
FROM jupyter/base-notebook
COPY week1 /home/$NB_USER/work
```

Create the `Dockerfile` file in `jupyter_solutions` directory, and paste the above two commands into it.

Build the image with command:

```
docker build -t jupyter_solutions .
```

After the image is built, you can check that this image added one layer created by the `COPY` command in `Dockerfile` using `docker image ls` to get the image ID, and `docker image <image_id>`. The size of this layer is 2MB.

You can start running it with the following command

```
docker run -it  --rm -p 8899:8888  jupyter_solutions
```

Again, you can find out the jupyter notebook's URL from the output message and load that in your browser. This time you will see a folder `week1` inside the `work` folder. You should be able to run `homework_week1.ipynb` to produce the intended results.

Question 5: Process management

In this exercise, we will see a little bit of the internals of docker. In particular, the process namespace. We need an interactive linux container to demonstrate that. We will use a `ubuntu` image.

Run the following command to start an interactive ubuntu container

```
docker run -it ubuntu
```

The docker daemon will pull the latest ubuntu image from DockerHub and start a containter based on the image. After the container starts, you will enter the container's `ubuntu` terminal. The only difference would be the prompt, which becomes something like `root@bb60e5736f7e`. The image gives you a terminal and you are 'logged in' as the root user.

Run `ps -ef` command to check all processes running in the container, you are likely to see an output like

```
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 00:24 pts/0    00:00:00 /bin/bash
root        19     1  0 00:35 pts/0    00:00:00 ps -ef
```

There are only two processes, the "init" process is started by command `/bin/bash`, our entry point to the container. It has a child process `ps -ef`.

Now run the following command to check the content of the current directory every 60 seconds. The process is instructed to run in the background so we can still use the terminal:

```
watch -n 60 ls -l &
```

Run `ps -ef` command again, you will find there are three processes running in the container. The output would be similar to the following:

```
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 00:24 pts/0    00:00:00 /bin/bash
root        20     1  0 00:35 pts/0    00:00:00 watch -n 60 ls -l
root        34     1  0 00:38 pts/0    00:00:00 ps -ef
```

The processes running inside the container is visible to the host operating system. Open another terminal on the host OS and type the command to find this process

```
ps -ef | grep watch
```

You will see a process started by command `watch -n 60 ls -l`, with a very different PID and PPID. Those are the PIDs assigned to the same process by the host OS. You are able to kill the process from the host OS. For instance if the PID of the `watch` process is `32332`. Using `sudo kill -9 32332` would kill the process. You can use the `ps -ef` command within the container to check that it is killed.

References

- Docker Getting Started. `https://docs.docker.com/storage/volumes/`
- Manage data in Docker. `https://docs.docker.com/storage/`
- Using Volumes. `https://docs.docker.com/storage/volumes/`
- Jupyter Base Notebook. `https://hub.docker.com/r/jupyter/base-notebook/`