Labwork 2:

Virtualization with Docker Engine – OS level virtualization

In this labwork you will learn to:

- Run docker engine on Windows 10
- Install and manage containers using docker engine from the registry
- Use the client interface command line.
- The reference of this lab can be found at https://docker-curriculum.com/

1. The first step is to enable the Intel Virtualization Technology or AMD-V from the BIOS. To enter the BIOS, restart your computer, and before the windows starts press F1, F2, F10, or F12, depending on your computer.
2. Download the docker engine on your computer from docker hub (hub.docker.com) and install.
3. Once install, start docker application, and run a PowerShell as administrator.
4. In your Power Shell window run the following commands

   ```
   Get-WindowsOptionalFeature -Online -FeatureName containers

   Enable-WindowsOptionalFeature -Online -FeatureName containers –All

   Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V –All
   ```

   What is the role of each elements of the above commands?

5. Once you are done installing Docker, test your Docker installation by running the following:

   `docker pull hello-world` : this command pull the hello-world image from the docker registry

   `docker run hello-world` : this command run the downloaded image in a container, *you can notice how fast it is to run a container with an OS inside*. What is the output after this operation?

6. `docker run busybox` (You might need to Switch to Linux Containers and in case the problem persists, enable experimental features from the settings) *When you call run, the Docker client finds the image (busybox in this case), loads up the container and then runs a command in that container. When we run docker run busybox, we didn't provide a command, so the container booted up, ran an empty command and then exited.*
7. `docker run busybox echo "hello from busybox"`: *Here the Docker client dutifully ran the echo command in our busybox container and then exited it. If you've noticed, all of that happened pretty quickly. Imagine booting up a virtual machine, running a command and then killing it.* **Now you know why they say containers are fast***!*
8. `docker ps` : *The docker ps command shows you all containers that are currently running.*
9. `docker ps -a` : *See all containers currently running and exited*.
10. `docker run -it busybox`: *Running the run command with the -it flags attaches us to an interactive tty in the container. Now we can run as many commands in the container as we want.*
11. `docker rm "machine ID"`: *as a rule of thumb, I clean up containers once I'm done with them. To do that, you can run the docker rm command.* **Just copy the container IDs from above and paste them alongside the command.**
12. `docker rm $(docker ps -a -q -f status=exited)`: the -q flag, only returns the numeric IDs and -f filters output based on conditions provided, --rm flag that can be passed to docker run which automatically deletes the container once it's exited from.
13. `docker container prune` : *This will remove all stopped containers*.

Terminology

In the last section, we used a lot of Docker-specific jargon which might be confusing to some. So before we go further, let me clarify some terminology that is used frequently in the Docker ecosystem.

- *Images* - The blueprints of our application which form the basis of containers. In the demo above, we used the `docker pull` command to download the **busybox** image.
- *Containers* - Created from Docker images and run the actual application. We create a container using `docker run` which we did using the busybox image that we downloaded. A list of running containers can be seen using the `docker ps` command.
- *Docker Daemon* - The background service running on the host that manages building, running and distributing Docker containers. The daemon is the process that runs in the operating system which clients talk to.
- *Docker Client* - The command line tool that allows the user to interact with the daemon. More generally, there can be other forms of clients too - such as Kitematic which provide a GUI to the users.
- *Docker Hub* - A registry of Docker images. You can think of the registry as a directory of all available Docker images. If required, one can host their own Docker registries and can use them for pulling images.

WEBAPPS with Docker: deploying web applications with Docker

We're going to pull a Docker image from Docker Hub, run the container and see how easy it is to run a webserver.

The image that we are going to use is a single-page website that is already created for the purpose of this demo and hosted on the **registry** - `prakhar1989/static-site`. We can **download** and **run** the **image** directly in one go using docker run. As noted above, the `--rm` **flag** automatically removes the **container** when it exits. The command to use is:

```
docker run --rm prakhar1989/static-site
```

the client will first fetch the image from the registry and then run the image.

Now that the server is running, how to see the website? What port is it running on? And more importantly, how do we access the container directly from our host machine? Hit Ctrl+C to stop the container.

Well, in this case, the client is not exposing any ports so we need to re-run the docker run command to publish ports. While we're at it, we should also find a way so that our terminal is not attached to the running container. This way, you can happily close your terminal and keep the container running. This is called **detached** mode. To this end, run the following command

```
docker run -d -P --name static-site prakhar1989/static-site
```

In the above command, -d will detach our terminal, -P will publish all exposed ports to random ports and finally --name corresponds to a name we want to give. Now we can see the ports by running the docker port [CONTAINER] command

```
docker port static-site
```

You can open http://localhost:32769 in your browser.

You can also specify a custom port to which the client will forward connections to the container.

```
docker run -p 8888:80 prakhar1989/static-site
```

To stop a detached container, run docker stop by giving the container ID. In this case, we can use the name static-site we used to start the container.

```
docker stop static-site
```

```
docker stop "machine ID"
```

Creating your first image / test environment

Each image / container you're going to create must be created from a base image of OS. Say you're app runs on Ubuntu 14, so my base image is Ubuntu 14. To create an image we're going to run Ubuntu 14 first:

`docker run -t -i ubuntu:14.04` : *-t tells docker to run a terminal and –i to make it interactive, so we can setup our apps.*

Now let's install an application. I'll just install Apache2/PHP5 and deploy a 'hello world' PHP script for now.

```
apt-get update
apt-get install apache2 php5 nano curl
nano /var/www/html/index.php
rm /var/www/html/index.html
service apache2 start
```

Now to verify

```
curl http://localhost
```

OK, now my running container has my dummy app running. Let's save it.

First we exit from the container to host.

```
exit
```

Then we save the running container:

```
docker commit <containerID> myapp
```

Let's verify that our image is saved…

```
docker images
```

This will bring out all list of images you have.

You've already seen that our clean test app lies as an image, which is ready for launch any moment. You can launch more than one instances in seconds.

Let's launch our very first test instance:

```
docker run -t -i myapp
```

Once inside the container you can run/test your app. For now:

```
service apache2 start
curl http://localhost
```

At the same time, you can open another terminal and launch parallel containers. Since each container runs in isolation, so does your application instance, which makes it testing heaven. No interaction with system or with each other.