Cloud Technologies – Lab Exercise Docker and RabbitMQ

The purpose of this lab exercise is firstly to become familiar with Docker if you have not already used it. The second objective is to implement the simple RabbitMQ tutorial in a 'dockerised' environment.

Introduction to Docker

The information below is extracted from the Docker documentation at https://docs.docker.com/

Follow the instructions for installing the latest Docker Community Edition at https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/

There is a brief non-technical introduction to docker at https://docs.docker.com/get-started/ however the notes below give you more detail...

Run the command `sudo docker run hello-world` to confirm docker has installed correctly. To make sure you are using version 1.13 or higher `run docker --version` to check it out. Note if you did not add your user to the `docker` group using `sudo usermod -aG docker $USER` then you will have to use sudo for each docker command.

Docker allows you to run applications, worlds you create, inside containers. Running an application inside a container takes a single command: `docker run`.

```
docker run alpine:3.6 /bin/echo 'Hello world'
```

The `docker run` combination *runs* containers.

Next we specified an image: `alpine:3.6`. This is the source of the container we ran. Docker calls this an image. In this case we used a popular minimal Linux distribution docker image called Alpine which is less than 5 MB !

When you specify an image, Docker looks first for the image on your Docker host. If it can't find it then it downloads the image from the public image registry: Docker Hub.

Next we told Docker what command to run inside our new container: `/bin/echo 'Hello world'`

To run an interactive docker container session in alpine use :

```
docker run -t -i alpine:3.6 /bin/sh
```

What is the ip address that has been assigned to this container ?

A daemonized Hello world

```
docker run -d ubuntu:14.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"
```

This command executes `docker run` but this time we specified a flag: `-d`. The `-d` flag tells Docker to run the container and put it in the background, to daemonize it. Note the size of this docker image for Ubuntu 14.04 compared with the alpine image size – use `docker images`

We can use the container ID to see what's happening with our `hello world` daemon. The `docker ps` command queries the Docker daemon for information about all the running containers it knows about. Add the -a parameter to see ALL the containers.

To see what is going on look inside the container using the `docker logs` command – you can use the container name Docker assigned or the container id.

The `docker stop` command with either the container name Docker assigned or the container id tells Docker to politely stop the running container. If it succeeds it will return the name of the container it has just stopped.

Running a web application in Docker

Start a Python Flask application with a `docker run` command. We are using a pre-built app however the Getting Started tutorial on docker docs brings you through setting up a similar app building a Dockerfile.

```
$ docker run -d -P training/webapp python app.py
```

Review what the command did. You've specified two flags: `-d` and `-P`. You've already seen the `-d` flag which tells Docker to run the container in the background. The `-P` flag is new and tells Docker to map any required network ports inside our container to our host. This lets us view our web application.

You've specified an image: `training/webapp`. This image is a pre-built image you've created that contains a simple Python Flask web application.

Lastly, you've specified a command for our container to run: `python app.py`. This launches our web application.

Viewing our web application container

Now you can see your running container using the `docker ps` command.

```
$ docker ps -l
CONTAINER ID  IMAGE                   COMMAND       CREATED        STATUS        PORTS                      NAMES
bc533791f3f5  training/webapp:latest  python app.py 5 seconds ago  Up 2 seconds  0.0.0.0:49155->5000/tcp    nostalgic_morse
```

You can see you've specified a new flag, `-l`, for the `docker ps` command. This tells the `docker ps` command to return the details of the *last* container started.

We can see the same details we saw when we first Dockerized a container with one important addition in the `PORTS` column.

```
PORTS
0.0.0.0:49155->5000/tcp
```

When we passed the `-P` flag to the `docker run` command Docker mapped any ports exposed in our image to our host.

In this case Docker has exposed port 5000 (the default Python Flask port) on port 49155.

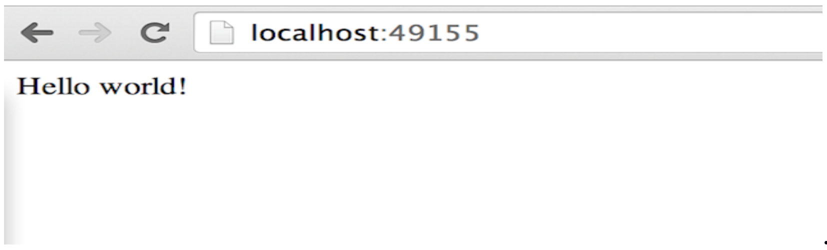
Network port bindings are very configurable in Docker. In our last example the `-P` flag is a shortcut for `-p 5000` that maps port 5000 inside the container to a high port (from *ephemeral port range* which typically ranges from 32768 to 61000) on the local Docker host. We can also bind Docker containers to specific ports using the `-p` flag, for example:

```
$ docker run -d -p 80:5000 training/webapp python app.py
```

This would map port 5000 inside our container to port 80 on our local host. You might be asking about now: why wouldn't we just want to always use 1:1 port mappings in Docker containers rather than mapping to high ports? Well 1:1 mappings have the constraint of only being able to map one of each port on your local host.

Suppose you want to test two Python applications: both bound to port 5000 inside their own containers. Without Docker's port mapping you could only access one at a time on the Docker host.

So you can now browse to port 49155 in a web browser to see the application.



Hello world!

Our Python application is live!

> **Note:** If you have been using a virtual machine on OS X, Windows or Linux, you'll need to get the IP of the virtual host instead of using localhost. You can do this by running the

`docker-machine ip your_vm_name` from your command line or terminal application, for example:

```
$ docker-machine ip my-docker-vm
192.168.99.100
```

In this case you'd browse to `http://192.168.99.100:49155` for the above example.

A network port shortcut

Using the `docker ps` command to return the mapped port is a bit clumsy so Docker has a useful shortcut we can use: `docker port`. To use `docker port` we specify the ID or name of our container and then the port for which we need the corresponding public-facing port.

```
$ docker port nostalgic_morse 5000
0.0.0.0:49155
```

In this case you've looked up what port is mapped externally to port 5000 inside the container.

Viewing the web application's logs

You can also find out a bit more about what's happening with our application and use another of the commands you've learned, `docker logs`.

```
$ docker logs -f nostalgic_morse
* Running on http://0.0.0.0:5000/
10.0.2.2 - - [23/May/2014 20:16:31] "GET / HTTP/1.1" 200 -
10.0.2.2 - - [23/May/2014 20:16:31] "GET /favicon.ico HTTP/1.1" 404 -
```

This time though you've added a new flag, `-f`. This causes the `docker logs` command to act like the `tail -f` command and watch the container's standard out. We can see here the logs from Flask showing the application running on port 5000 and the access log entries for it.

Looking at our web application container's processes

In addition to the container's logs we can also examine the processes running inside it using the `docker top` command.

```
$ docker top nostalgic_morse
PID                 USER                COMMAND
854                 root                python app.py
```

Here we can see our `python app.py` command is the only process running inside the container.

Inspecting our web application container

Lastly, we can take a low-level dive into our Docker container using the `docker inspect` command. It returns a JSON document containing useful configuration and status information for the specified container.

```
$ docker inspect nostalgic_morse
```

You can see a sample of that JSON output.

```
[{
    "ID": "bc533791f3f500b280a9626688bc79e342e3ea0d528efe3a86a51ecb28ea20",
    "Created": "2014-05-26T05:52:40.808952951Z",
    "Path": "python",
    "Args": [
       "app.py"
    ],
    "Config": {
       "Hostname": "bc533791f3f5",
       "Domainname": "",
       "User": "",
. . .
```

We can also narrow down the information we want to return by requesting a specific element, for example to return the container's IP address we would:

```
$ docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
nostalgic_morse
172.17.0.5
```

Stopping our web application container

Okay you've seen web application working. Now you can stop it using the `docker stop` command and the name of our container: `nostalgic_morse`.

```
$ docker stop nostalgic_morse
nostalgic_morse
```

We can now use the `docker ps` command to check if the container has been stopped.

```
$ docker ps -l
```

Restarting our web application container

Oops! Just after you stopped the container you get a call to say another developer needs the container back. From here you have two choices: you can create a new container or restart the old one. Look at starting your previous container back up.

```
$ docker start nostalgic_morse
nostalgic_morse
```

Now quickly run `docker ps -l` again to see the running container is back up or browse to the container's URL to see if the application responds.

> **Note:** Also available is the `docker restart` command that runs a stop and then start on the container.

Removing our web application container

Your colleague has let you know that they've now finished with the container and won't need it again. Now, you can remove it using the `docker rm` command.

```
$ docker rm nostalgic_morse
Error: Impossible to remove a running container, please stop it first or use -f
2014/05/24 08:12:56 Error: failed to remove one or more containers
```

What happened? We can't actually remove a running container. This protects you from accidentally removing a running container you might need. You can try this again by stopping the container first.

```
$ docker stop nostalgic_morse
nostalgic_morse
$ docker rm nostalgic_morse
nostalgic_morse
```

And now our container is stopped and deleted.

Listing images on the host

Let's start with listing the images you have locally on our host. You can do this using the `docker images` command like so:

```
sudo docker images
REPOSITORY TAG                IMAGE ID            CREATED             VIRTUAL SIZE
ubuntu     14.04              e9ae3c220b23        6 days ago          187.9 MB
ubuntu     latest             e9ae3c220b23        6 days ago          187.9 MB
python     2                  b8f0eed60dfc        3 weeks ago         675.1 MB
rabbitmq   3-management       2095dd2f4b13        3 weeks ago         182.9 MB
rabbitmq   3                  9da54b587274        3 weeks ago         182.9 MB
hello-world latest            0a6ba66e537a        4 weeks ago         960 B
training/webapp latest        54bb4e8718e8        6 months ago        348.8 MB
```

Each image has been downloaded from Docker Hub when you launched a container using that image. When you list images, you get three crucial pieces of information in the listing.

- What repository they came from, for example `ubuntu`.
- The tags for each image, for example `14.04`.
- The image ID of each image.

So how do you get new images? Well Docker will automatically download any image you use that isn't already present on the Docker host. But this can potentially add some time to the launch of a container. If you want to pre-load an image you can download it using the `docker pull` command. Suppose you'd like to download the `centos` image.

```
$ docker pull centos
Pulling repository centos
b7de3133ff98: Pulling dependent layers
5cc9e91966f7: Pulling fs layer
511136ea3c5a: Download complete
ef52fb1fe610: Download complete
```

```
. . .
Status: Downloaded newer image for centos
```

You can see that each layer of the image has been pulled down and now you can run a container from this image and you won't have to wait to download the image.

```
$ docker run -t -i centos /bin/bash
bash-4.1#
```

One of the features of Docker is that a lot of people have created Docker images for a variety of purposes. Many of these have been uploaded to Docker Hub. You can search these images on the Docker Hub website.

You can also search for images on the command line using the `docker search` command. Suppose your team wants an image with RabbitMQ installed on which to do our web application development. You can search for a suitable image by using the `docker search` command to find all the images that contain the term `RabbitMQ`

```
sudo docker search Rabbitmq
NAME                            DESCRIPTION                                     STARS     OFFICIAL
AUTOMATED
rabbitmq                        RabbitMQ is a highly reliable enterprise m...   291       [OK]
fedora/rabbitmq                                                                 18        [OK]
networld/rabbitmq               Networld PaaS RabbitMQ image in default in...   4         [OK]
gonkulatorlabs/rabbitmq         35MB RabbitMQ image running on Alpine Linux     3         [OK]
mikaelhg/docker-rabbitmq        RabbitMQ in Docker.                             3         [OK]
frodenas/rabbitmq               A Docker Image for RabbitMQ                     3         [OK]
…........
```

You've reviewed the images available to use and you decided to use the `networld/rabbitmq` image. So far you've seen two types of images repositories, images like `ubuntu`, which are called base or root images. These base images are provided by Docker Inc and are built, validated and supported. These can be identified by their single word names. A user image belongs to a member of the Docker community and is built and maintained by them. You can identify user images as they are always prefixed with the user name, here `networld`, of the user that created them.

Creating our own images

There are two ways you can update and create images.

1. You can update a container created from an image and commit the results to an image.
2. You can use a `Dockerfile` to specify instructions to create an image.

For details on how to do this see https://docs.docker.com/engine/tutorials/dockerimages/

Networking containers

You can name your container by using the `--name` flag, for example launch a new container called web:

```
$ docker run -d -P --name web training/webapp python app.py
```

You can also use `docker inspect` with the container's name. Container names must be unique. That means you can only call one container `web`. If you want to re-use a container name you must delete the old container (with `docker rm`) before you can reuse the name with a new container. Go ahead and stop and them remove your `web` container.

Docker includes support for networking containers through the use of **network drivers**. By default, Docker provides two network drivers for you, the `bridge` and the `overlay` driver.

Every installation of the Docker Engine automatically includes three default networks. You can list them:

```
$ docker network ls
NETWORK ID          NAME                DRIVER
18a2866682b8        none                null
c288470c46f6        host                host
7b369448dccb        bridge              bridge
```

The network named `bridge` is a special network. Unless you tell it otherwise, Docker always launches your containers in this network.

```
$ docker run -itd --name=networktest ubuntu
```

Inspecting the network is an easy way to find out the container's IP address.

```
$ docker inspect networktest
```

You can remove a container from a network by disconnecting the container.

```
$ docker network disconnect bridge networktest
```

Networks are natural ways to isolate containers from other containers or other networks.

Data volumes

A *data volume* is a specially-designated directory within one or more containers that bypasses the *Union File System*. Data volumes provide several useful features for persistent or shared data:

- Volumes are initialized when a container is created. If the container's base image contains data at the specified mount point, that existing data is copied into the new volume upon volume initialization.
- Data volumes can be shared and reused among containers.
- Changes to a data volume are made directly.
- Changes to a data volume will not be included when you update an image.
- Data volumes persist even if the container itself is deleted.

Data volumes are designed to persist data, independent of the container's life cycle. Docker therefore *never* automatically deletes volumes when you remove a container, nor will it "garbage collect" volumes that are no longer referenced by a container.

Adding a data volume

You can add a data volume to a container using the `-v` flag with the `docker create` and `docker run` command. You can use the `-v` multiple times to mount multiple data volumes. Let's mount a single volume now in our web application container.

```
$ docker run -d -P --name web -v /webapp training/webapp python app.py
```

This will create a new volume inside a container at `/webapp`.

Docker volumes default to mount in read-write mode, but you can also set it to be mounted read-only.

```
$ docker run -d -P --name web -v /opt/webapp:ro training/webapp python app.py
```

Locating a volume

You can locate the volume on the host by utilizing the 'docker inspect' command.

```
$ docker inspect web
```

The output will provide details on the container configurations including the volumes. The output should look something similar to the following:

```
...
Mounts": [
    {
        "Name": "fac362...80535",
        "Source": "/var/lib/docker/volumes/fac362...80535/_data",
        "Destination": "/webapp",
        "Driver": "local",
        "Mode": "",
        "RW": true
    }
]
...
```

You will notice in the above 'Source' is specifying the location on the host and 'Destination' is specifying the volume location inside the container. `RW` shows if the volume is read/write.

Mount a host directory as a data volume

In addition to creating a volume using the `-v` flag you can also mount a directory from your Docker daemon's host into a container.

```
$ docker run -d -P --name web -v /src/webapp:/opt/webapp training/webapp python
app.py
```

This command mounts the host directory, `/src/webapp`, into the container at `/opt/webapp`. If the path `/opt/webapp` already exists inside the container's image, the `/src/webapp` mount overlays but does not remove the pre-existing content. Once the mount is removed, the content is accessible again. This is consistent with the expected behavior of the `mount` command.

The `container-dir` must always be an absolute path such as `/src/docs`. The `host-dir` can either be an absolute path or a `name` value. If you supply an absolute path for the `host-dir`, Docker bind-mounts to the path you specify. If you supply a `name`, Docker creates a named volume by that `name`.

A `name` value must start with start with an alphanumeric character, followed by `a-z0-9`, `_` (underscore), `.` (period) or `-` (hyphen). An absolute path starts with a `/` (forward slash).

For example, you can specify either `/foo` or `foo` for a `host-dir` value. If you supply the `/foo` value, Docker creates a bind-mount. If you supply the `foo` specification, Docker creates a named volume.

Docker commands and Docker Hub

The Docker Hub is a public registry maintained by Docker, Inc. It contains images you can download and use to build containers. It also provides authentication, work group structure, workflow tools like webhooks and build triggers, and privacy tools like private repositories for storing images you don't want to share publicly.

Docker itself provides access to Docker Hub services via the `docker search`, `pull`, `login`, and `push` commands.

Docker Compose and Docker Swarm

A `docker-compose.yml` file is a YAML file that defines how Docker containers should behave in production. In a distributed application, different pieces of the app are called "services." Look at https://docs.docker.com/get-started/part3/ to see how to build services to scale the application and enable load balancing.

If you want to work with Docker swarms and use the docker machine to create swarm nodes then you can work through Part 4 of the Docker Getting started tutorial https://docs.docker.com/get-started/part4/ where you will deploy an application onto a cluster, running it on multiple machines. Multi-container, multi-machine applications are made possible by joining multiple machines into a "Dockerized" cluster called a swarm.

A swarm is a group of machines that are running Docker and joined into a cluster. After that has happened, you continue to run the Docker commands you're used to, but now they are executed on a cluster by a **swarm manager**. The machines in a swarm can be physical or virtual. After joining a swarm, they are referred to as **nodes**.

Swarm managers can use several strategies to run containers, such as "emptiest node" – which fills the least utilized machines with containers. Or "global", which ensures that each machine gets exactly one instance of the specified container. You instruct the swarm manager to use these strategies in the Compose file, just like the one you have already been using.

Swarm managers are the only machines in a swarm that can execute your commands, or authorize other machines to join the swarm as **workers**. Workers are just there to provide capacity and do not have the authority to tell any other machine what it can and cannot do.

What is RabbitMQ?

RabbitMQ is open source message broker software (sometimes called message-oriented middleware) that implements the Advanced Message Queuing Protocol (AMQP). The RabbitMQ server is written in the Erlang programming language and is built on the Open Telecom Platform framework for clustering and failover. Client libraries to interface with the broker are available for all major programming languages.

You should follow the simple python Hello World tutorial at http://www.rabbitmq.com/getstarted.html

Exercise :

Your exercise is to 'dockerise' this simple Hello World program.

One of the important things to note about RabbitMQ is that it stores data based on what it calls the "Node Name", which defaults to the hostname. What this means for usage in Docker is that we should either specify `-h/--hostname` or `-e RABBITMQ_NODENAME=...` explicitly for each daemon so that we don't get a random hostname and can keep track of our data:

```
docker run -d -e RABBITMQ_NODENAME=my-rabbit --name some-rabbit rabbitmq:3
```

If you give that a minute, then do `docker logs some-rabbit`, you'll see in the output a block similar to:

```
sudo docker logs some-rabbit2

              RabbitMQ 3.5.6. Copyright (C) 2007-2015 Pivotal Software, Inc.
  ##  ##      Licensed under the MPL.  See http://www.rabbitmq.com/
  ##  ##
  ##########  Logs: tty
  ######  ##        tty
  ##########
              Starting broker...
=INFO REPORT==== 11-Nov-2015::10:10:51 ===
Starting RabbitMQ 3.5.6 on Erlang 18.1
Copyright (C) 2007-2015 Pivotal Software, Inc.
Licensed under the MPL.  See http://www.rabbitmq.com/

=INFO REPORT==== 11-Nov-2015::10:10:51 ===
node           : rabbit@my-rabbit
home dir       : /var/lib/rabbitmq
config file(s) : /etc/rabbitmq/rabbitmq.config
cookie hash    : kLTVJEpsd55efsdfSiHn2w==
log            : tty
sasl log       : tty
```

```
database dir   : /var/lib/rabbitmq/mnesia/rabbit@my-rabbit
```

Note the `database dir` there, especially that it has my `RABBITMQ_NODENAME` appended to the end for the file storage. This image makes all of `/var/lib/rabbitmq` a volume by default.

Management Plugin

There is a second set of tags provided with the management plugin installed and enabled by default, which is available on the standard management port of 15672, with the default username and password of `guest` / `guest`:

```
docker run -d -e RABBITMQ_NODENAME=my-rabbit --name some-rabbit rabbitmq:3-management
```

You can access it by visiting `http://container-ip:15672` in a browser or, if you need access outside the host, on port 8080:

```
docker run -d -e RABBITMQ_NODENAME=my-rabbit --name some-rabbit -p 8080:15672
rabbitmq:3-management
```

You can then go to `http://localhost:8080` or `http://host-ip:8080` in a browser.

Username and password both guest.

Connecting to the daemon

```
docker run --name some-app --link some-rabbit:rabbit -d application-that-uses-
rabbitmq
```

You should download and save the two python files send.py and receive.py. You will need to change the 'localhost' ip address in these files to point to the ip address of the container in which you have the RabbitMQ server process running.

The following command should run a python container, linking it to the container running the RabbitMQ server and also allowing access to the send/receive python files in you current working directory.

```
 sudo docker run -v $PWD:/code --name send --link=some-rabbit -it python:2 bash
 sudo docker run -v $PWD:/code --name receive --link=some-rabbit -it python:2 bash
```

In order to run the python files you will need to install the pika module (you can use pip install pika).

Finally you should be able to run the send and receive programs which will communicate using the RabbitMQ service running in your container.
You will need to create your own Docker images for the send and receive programs.

To complete the exercise you should push your customised docker instances to Docker Hub.