Exercise 4

Using Docker

Prior Knowledge
Unix command-line
Apt package manager
Amazon AWS / EC2 Console

Learning Objectives
Be able to instantiate docker containers
Be able to modify docker containers and save them
Interacting with the docker hub
Creating a dockerfile
Using Docker Compose

Software Requirements

- Docker
- Docker-Compose
- Ubuntu
- Visual Studio Code

Introduction
This lab consists of three parts. The first part is just playing around with Docker to understand how stuff works. The things we are going to do are not typical docker usage as we are investigating the way the system works

The second part involves creating a dockerfile which is a sort of build file. This is the more usual usage of Docker and will stand you in good stead for many projects.

Finally we will load your newly created docker image up in EC2.

PART A – understanding the Docker model

1. Let's start by running a CentOs image inside our Ubuntu VM.

2. From the Ubuntu command-line, type:
   `docker pull centos`

3. You should see something like:

```
Using default tag: latest
latest: Pulling from library/centos
7a0437f04f83: Pull complete
Digest:
sha256:5528e8b1b1719d34604c87e11dcd1c0a20bedf46e83b5632cdeac91b8c04efc1
Status: Downloaded newer image for centos:latest
```

4. We will take a look at what this means shortly, but first lets try it out.
   `docker run -ti centos /bin/bash`

   Hint:
   -ti basically means run this container in interactive mode. For more explanation see: https://docs.docker.com/engine/reference/run/

   You should see:
   `[root@c22c9c908236 /]#`

   Did you notice how fast it started?! This is not your usual VM.

   Let's refer to this window as the *docker window.*

5. Now type
   `ls /home/oxsoa`

   This will fail, because we are now in a mini virtual machine. Now try

```
apt-get
```

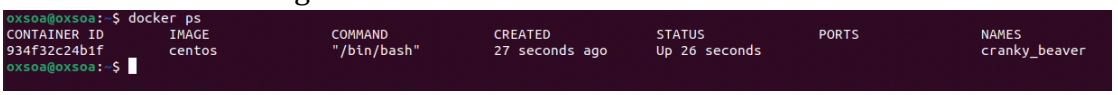
Again it fails. But what about `yum`?

Why does yum succeed? Because yum is the package manager for CentOS and now we are in a CentOS world. (Actually we won't use yum or apt-get *within* the docker… we'll come to how that works shortly).

6. Start a separate window. Let's refer to this as the *control window.* Now type

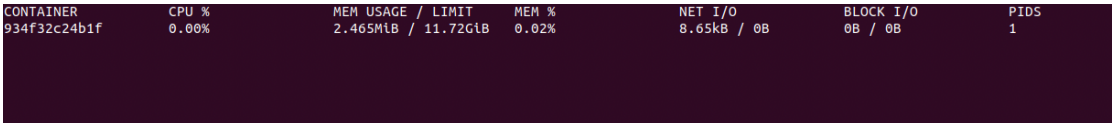
   ```
   docker ps
   ```

7. You will see something like:



8. Docker has given your container instance a random name (in my case cranky_beaver). You can now see how this instance is doing:

   ```
   docker stats cranky_beaver
   ```
   Obviously change *cranky_beaver* to the name of your container!



9. Notice how little memory each container takes. This means you can run hundreds of containers on a normal machine.

10. Now **Ctrl-C** to exit that command.

11. Now go onto http://hub.docker.com and signup. You need a valid email address to complete signup. I think you might want to do this in your own name because it's a useful system.

12. Once you have signed up, then do a docker login:
    `docker login -u` *`yourdockerhubuserid`*

13. Back in the control window, type

    ```
    docker commit <your_container_name> <yr_dock_id>/mycentos
    ```
    e.g. for me that would be

    ```
    docker commit cranky_beaver pizak/mycentos
    ```

14. Now list the images you have locally

```
docker images
```

15. You will see something like:

```
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
pizak/mycentos      latest              9f154062124f        21 minutes ago      172.3 MB
centos              latest              ce20c473cd8a        5 weeks ago         172.3 MB
```

16. Actually it would be useful to give that image a version name:

```
docker tag yourdocker/mycentos yourdocker/mycentos:1
```

17. Repeat the "docker images" command.

18. Now let's push that image up to the docker hub:

```
docker push yourdocker/mycentos:1
```

Enter your docker hub credentials if prompted.

19. The system will whirr away and upload some stuff. Eventually you will see something like:

```
The push refers to a repository [pizak/mycentos] (len: 1)
9f154062124f: Image already exists
ce20c473cd8a: Image successfully pushed
4234bfdd88f8: Image already exists
812e9d9d677f: Image already exists
168a69b62202: Image successfully pushed
47d44cb6f252: Image already exists
Digest:
sha256:f751347496258e359fdc065b468ff7d72302cbb6f2310adee802b6c5ff92615d
```

20. Now let's go back to the original docker window, where your image is still running. Make a new file in home like this:

```
[root@482fe4e23a8b /]# cd home
[root@482fe4e23a8b home]# echo hi > hi
[root@482fe4e23a8b home]# ls
hi
```

21. Now in your control terminal you can commit this change:

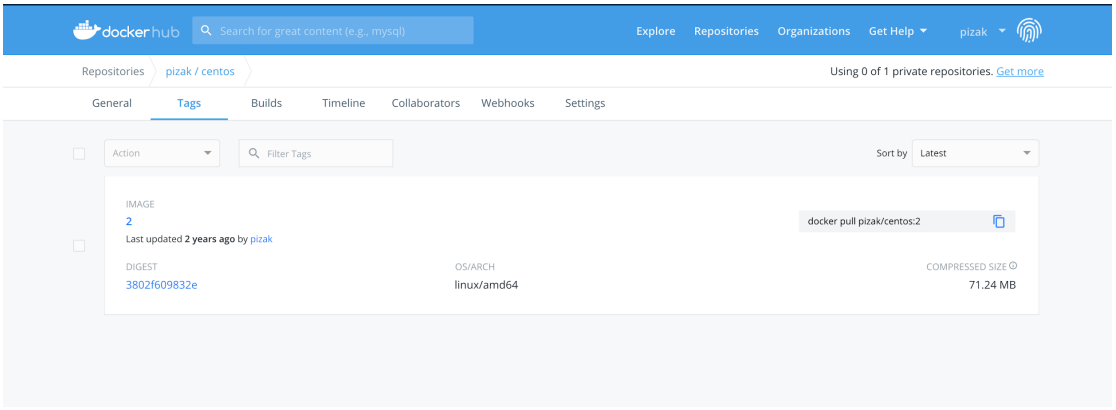
```
docker commit cranky_beaver yourdocker/mycentos:2
```

22. Let's push that image you've just made up to the Docker hub:

```
docker push yourdocker/mycentos:2
```

23. Notice how this time only a few bytes were uploaded. This is because of the layered file-system that docker uses to only save incremental changes. It is one of the major benefits of the docker system.

24. Go to the docker website http://hub.docker.com and view your repositories. In particular look at the tags tab:



25. You can now pull this docker image and create a container anywhere you like. Let's try some stuff out. From your *docker window* first exit the container by typing exit or Ctrl-D.

26. Now let's start v1 of your container:

    ```
    docker run -ti yourdocker/mycentos:1 /bin/bash
    ```

    Try looking at the home directory:
    `ls /home`

Now exit and load version 2

```
docker run -ti yourdocker/mycentos:2 /bin/bash
ls /home
```

27. Exit that container as well.

28. To prove that this is saved in the docker repo, do the following:

    First delete all the images locally that were tagged with your userid: *(Replace yourdocker with your userid)*

    ```
    docker rmi -f  $(docker images -q yourdocker/*)
    ```

29. Now try to start v1 and then v2 again. You will see that docker automatically re-downloads this and then runs it. Check that your file exists in the /home directory. Notice how fast the start up is when we already have the centos image but not the layers on top of it.

30. The one thing we haven't yet seen is how to get a docker image to do something vaguely useful.

31. First check you have nothing running locally on port 80. Browse to http://localhost:80 It should fail.

32. Now in your docker window, type:

```
docker run -p 80:80 httpd
```

33. You should see a bunch of stuff like this:

```
docker run -d -p 80:80 httpd
Unable to find image 'httpd:latest' locally
latest: Pulling from library/httpd
6f28985ad184: Pull complete
3a141a09d1d0: Pull complete
1633384edb75: Pull complete
acb3e3b931b8: Pull complete
f6dc6b8b1d70: Pull complete
Digest: sha256:9625118824bc2514d4301b387c091fe802dd9e08da7dd9f44d93ee65497e7c1c
Status: Downloaded newer image for httpd:latest
D01c921344d3244654df3ac0e179e92a465486bea6f881348ed7318c8e18ab56
AH00558: httpd: Could not reliably determine the server's fully qualified
domain name, using 172.17.0.2. Set the 'ServerName' directive globally to
suppress this message
AH00558: httpd: Could not reliably determine the server's fully qualified
domain name, using 172.17.0.2. Set the 'ServerName' directive globally to
suppress this message
[Mon Mar 22 06:22:44.029804 2021] [mpm_event:notice] [pid 1:tid
139781260461184] AH00489: Apache/2.4.46 (Unix) configured -- resuming normal
operations
[Mon Mar 22 06:22:44.029971 2021] [core:notice] [pid 1:tid 139781260461184]
AH00094: Command line: 'httpd -D FOREGROUND'
```

34. Now browse http://localhost:80 again and you should see.

35. *Are you wondering what –p 80:80 means?*
    It means expose port 80 from within the container as port 80 in the host system.

36. Now kill that container (Ctrl-C) and start it again in detached mode.
    This is how you would normally run a docker workload.

```
docker run -d -p 80:80 httpd
```

37. Test http://localhost again

38. To find your docker runtime try

```
docker ps

CONTAINER ID        IMAGE               COMMAND              CREATED
STATUS              PORTS               NAMES
f9ed00d6c251        httpd:latest        "httpd-foreground"   5 seconds
      ago       Up 4 seconds        0.0.0.0:80->80/tcp
      reverent_lalande
```

and finally to stop it

```
docker kill reverent_lalande
```


Recap:
In this section we have learnt basic docker commands including run, ps, image, commit, push and pull. We have learnt about the layered file system, and also about the docker repository.
We have looked at exposing network ports, how to start detached workloads and how to kill them.

In particular, notice how the docker containers seem like processes, but with the complete configuration neatly packaged and contained within a single packaged system that can be versioned, pushed and pulled. This model is ideal for creating and managing *microservices.*

PART B – Building a container using a Dockerfile

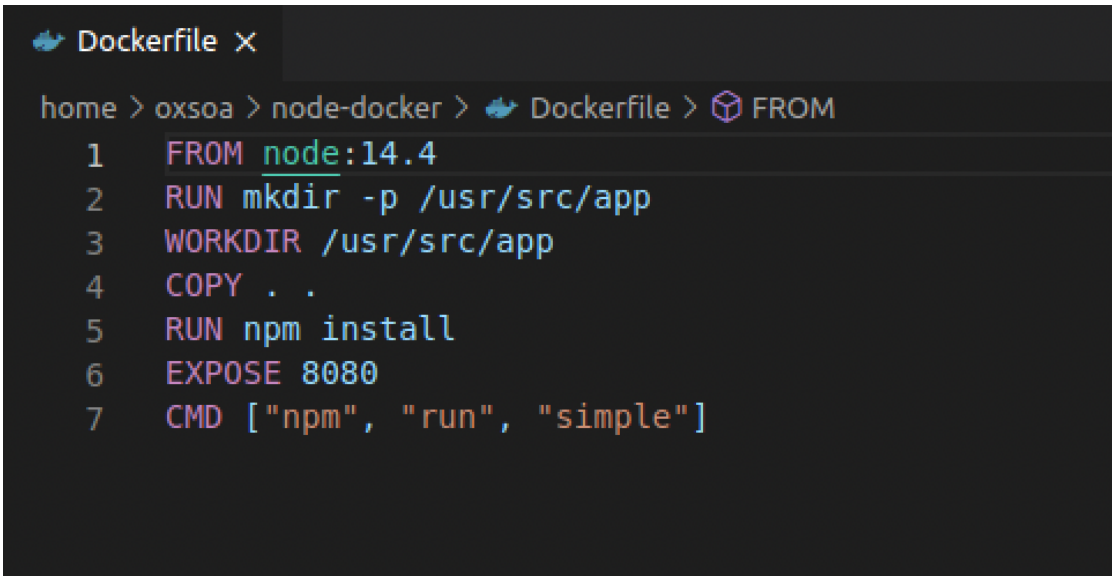
39. While I can imagine it might be possible to create docker images by modifying them like we have and then saving them, this is not a repeatable easy to use approach. Instead we want to build a dockerfile in a repeatable way.

40. Clone the git repository:
    ```
    git clone https://github.com/pzfreo/node-docker.git
    ```

41. Then
    ```
    cd node-docker
    ```


42. Take a look at the Dockerfile
    `code Dockerfile`

    It should look like:



    What this does is as follows:
    a) Start with existing Docker image called node:14.4 (which is the official release of node.js as a Docker image).
    b) Make a directory for our code
    c) Set that as the working directory
    d) Copy the source code over
    e) Install the dependencies needed to run the node app
    f) Tell docker that this listens on port 8080
    g) Use "npm run simple" as the executable command for the container

43. Now
    `docker build -t <your_docker_id>/nodeapp:1 .`

    (notice the '.'!)

44. While it is building, take a look at the docker file and also the reference guide:
    https://docs.docker.com/engine/reference/builder/

45. Once it has built, try running it:

    ```
    docker run --name nodeapp -d -p 80:8080  <yrdockerid>/nodeapp:1
    ```

46. Use a command-line HTTP tool:
    `curl -v http://localhost`

    You should see:

```
*   Trying 127.0.0.1:80...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 80 (#0)
> GET / HTTP/1.1
> Host: localhost
> User-Agent: curl/7.68.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: application/json; charset=utf-8
< Content-Length: 17
< ETag: W/"11-bDqgrL9BMdXEel/cuhi4kqHYo8U"
< Date: Sun, 28 Jun 2020 18:09:10 GMT
< Connection: keep-alive
<
* Connection #0 to host localhost left intact
{"a":"1","b":"2"}
```

47. Kill that container:

```
docker rm --force nodeapp
```

48. Ok. We have successfully run a simple server. However, we really would like to run our complete server that queries data including the database.

    You might think we would create a docker container containing both the server AND the mysql database. No! In Docker we basically have a process per container.

49. So to make our app work, we need to run two containers:
    a. node and
    b. mysql

50. Docker has a way of doing just that called docker-compose. In the lectures we will look at Kubernetes that does a LOT more than this.

51. Look at docker-compose.yaml
    `code docker-compose.yaml`

docker-compose.yaml

home > oxclo > node-docker > docker-compose.yaml

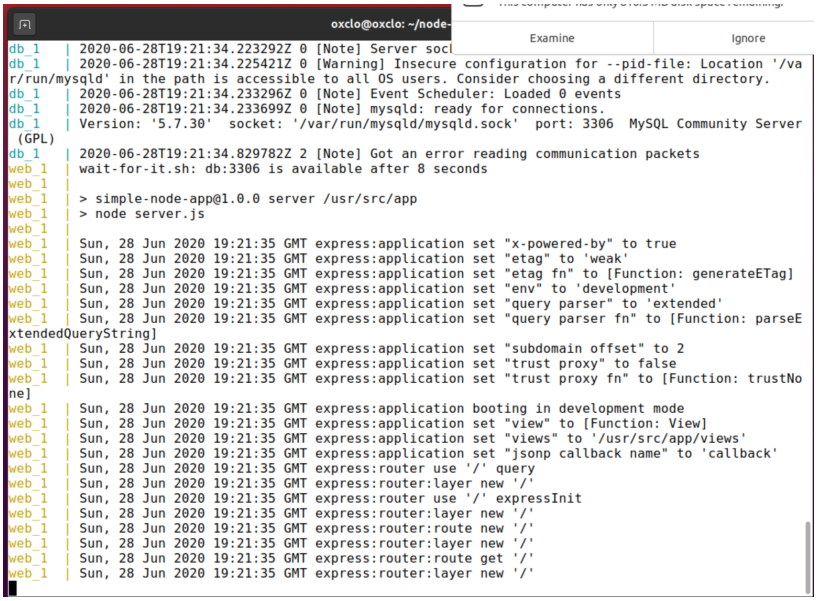
```yaml
version: '3'
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile.node
    image: nodejs
    depends_on:
      - db
    ports:
      - "80:8080"
    restart: unless-stopped
    command: ["./wait-for-it.sh", "db:3306", "--","npm", "run","server"]
    networks:
      - backend
    environment:
      DEBUG: "*:*"
      DBUSER: "root"
      DBPW: "secret"
      DBHOST: "db"
      DBNAME: "oxclo"
  db:
    build:
      context: .
      dockerfile: Dockerfile.mysql
    restart: always
    environment:
      MYSQL_DATABASE: "oxclo"
      MYSQL_ROOT_PASSWORD: "secret"
    networks:
      - backend
networks:
  backend:
    driver: bridge
```

I won't explain everything, but here are some key points:

a. We have two "services" which will be container runtimes: "web" and "db"
b. Web depends on db, so won't start until the other container is started
c. However, mysql takes some time to start up, so we need a little utility called "wait-for-it.sh" which waits until port 3306 is ready on the db container before letting the node app start.
d. We configure everything through environment variables, especially the links between the containers
e. There is a "virtual" bridge network that the two container runtimes use to communicate. Notice that we are binding the web container's ports to the outside world (mapping 80 to 8080) but we are not exposing port 3306. Therefore the database is only accessible by the web container.
f. Rather than use the default container image for mysql, we have extended it using Dockerfile.mysql - take a look at that file and the directory sql_scripts as well

52. Now let's start the "composition"
    `docker-compose up --build`

53. You should see lots of logging go by until finally you see:



54. In another window try:

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

You should see successful query of the database

You can clear up by typing hitting **Ctrl-C** to stop the composition. Then (in the `node-docker` directory!)

```
docker-compose down
```

Congratulations, you have completed the exercise.

Extension

If you have a github account, you can put the Dockerfile into the repository and automatically build it. Have a go.

Some rough hints:

Fork my node-docker repo into your github
In http://hub.docker.com go to Settings (click on your username)
Choose Linked Accounts and Services
Link to your Github account. Choose the "Public and Private"
Now click on Create (next to search) and Create Automated Build.
Select your github repository.
Enter a description. Click Create.
Now check the build status in the Build details tab. It takes about 3 minutes to build. If it is not building you can manually trigger it from the build settings.
Try doing an update to the dockerfile (maybe a spare comment) and then git push.