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
Running a docker container on EC2

Software Requirements

- AWS
- Docker
- Docker-Machine
- Docker-Compose
- Ubuntu
- Nano text editor

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:

```
latest: Pulling from centos
47d44cb6f252: Pull complete
168a69b62202: Pull complete
812e9d9d677f: Pull complete
4234bfdd88f8: Pull complete
ce20c473cd8a: Pull complete
centos:latest: The image you are pulling has been verified.
Important: image verification is a tech preview feature and
should not be relied on to provide security.
Digest:
sha256:c96eeb93f2590858b9e1396e808d817fa0ba4076c68b59395445cb957b
524408
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/oxclo`

   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).

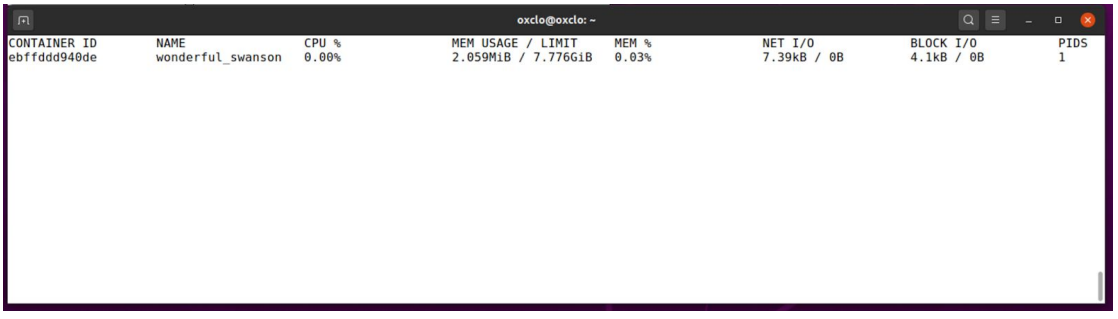
6. Start a separate window. Let's refer to this as the *control window.* Now type

   `docker ps`

7. You will see something like:

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

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



9. Notice how little memory each container takes.

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.

    ```
    docker commit drunk_engelbart 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 pizak/mycentos pizak/mycentos:1
    ```

17. Repeat the "docker images" command.

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

    ```
    docker push pizak/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:

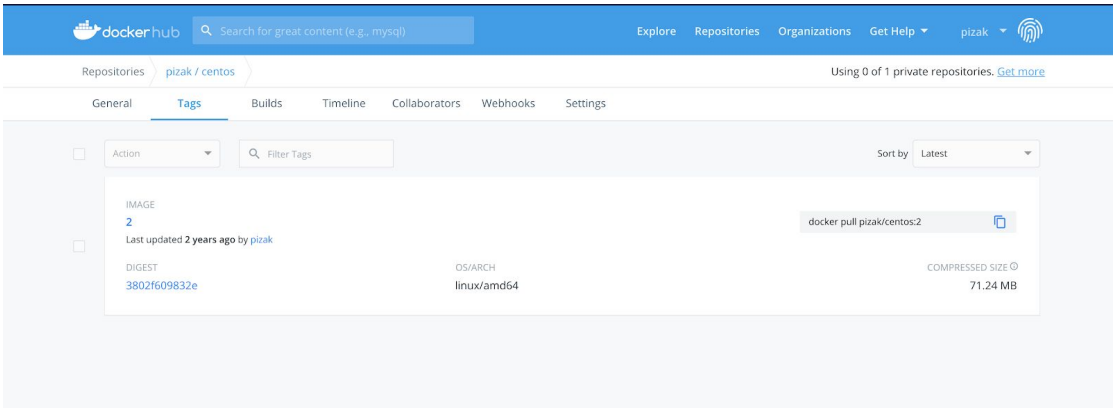
```
docker commit drunk_engelbart pizak/mycentos:2
```

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

```
docker push pizak/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 pizak/mycentos:1 /bin/bash
```

Try looking at the home directory:

```
ls /home
```

Now exit and load version 2

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

27. 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 pizak with your userid)*

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

28. Now try to start v2 again. You will see that docker automatically re-downloads this and then runs it. Check that your file exists in the /home directory.

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

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

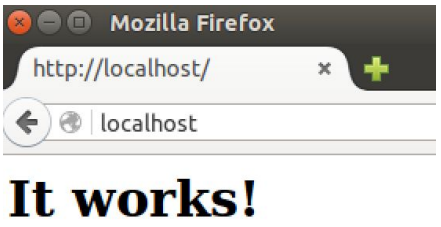
31. Now in your docker window, type:

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

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

```
Unable to find image 'httpd:latest' locally
latest: Pulling from httpd
ef2704e74ecc: Pull complete
1d6f63d023f5: Pull complete
…
781a5fd1cabf: Pull complete
bbd8adcb3ad5: Pull complete
6f953eead92f: Pull complete
afa235ca0577: Pull complete
f6d0a9cc3857: Pull complete
3fdd2b382f43: Pull complete
httpd:latest: The image you are pulling has been verified. Important:
image verification is a tech preview feature and should not be relied
on to provide security.
Digest:
sha256:fe40d6cb973ad7acbbc5fa99867efc03474649250a54da002fddaa88c6a5ff2f
Status: Downloaded newer image for httpd:latest
AH00558: httpd: Could not reliably determine the server's fully
qualified domain name, using 172.17.0.11. Set the 'ServerName'
directive globally to suppress this message
AH00558: httpd: Could not reliably determine the server's fully
qualified domain name, using 172.17.0.11. Set the 'ServerName'
directive globally to suppress this message
[Fri Nov 20 14:08:08.239803 2015] [mpm_event:notice] [pid 1:tid
140576655767424] AH00489: Apache/2.4.17 (Unix) configured -- resuming
normal operations
[Fri Nov 20 14:08:08.239940 2015] [core:notice] [pid 1:tid
140576655767424] AH00094: Command line: 'httpd -D FOREGROUND'
```


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



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

35. 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
```

36. Test http://localhost again

37. 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

38. 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.

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

40. Then
    ```
    cd node-docker
    ```


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

    It should look like:

```
FROM node:14.4
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY . .
RUN npm install
EXPOSE 8080
CMD ["npm", "run", "simple"]
```

    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

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

    (notice the '.'!)

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

44. Once it has built, try running it:

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


45. 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"}
```

46. Kill that container:

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

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

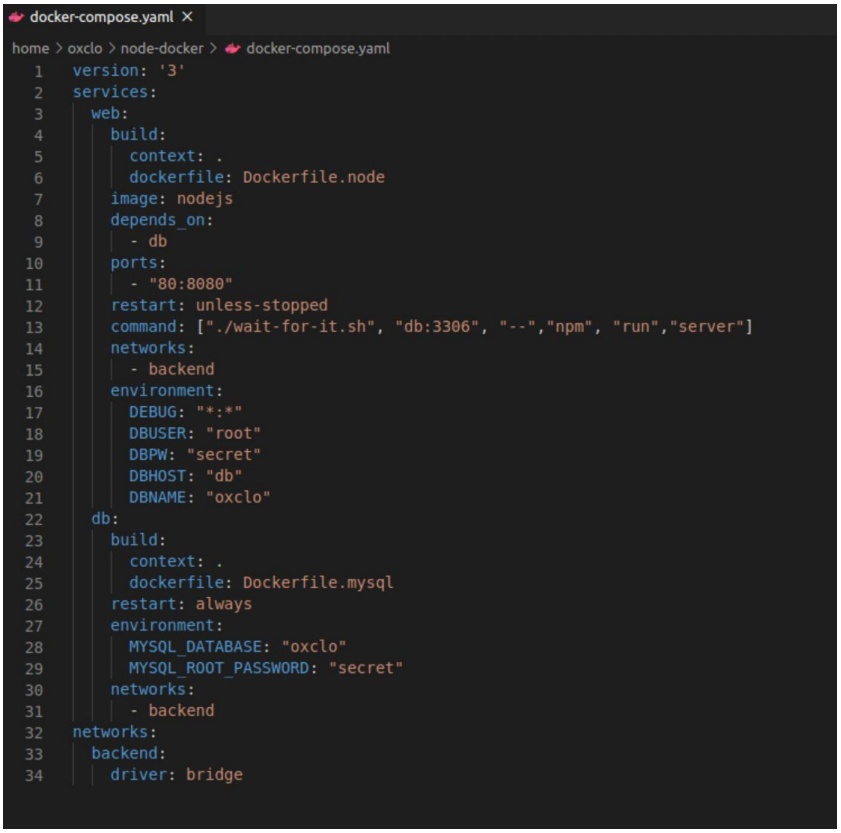
    You might think we would create a docker container containing both the server AND the mysql database. No! In Docker we basically have a process per container.

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

49. 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
```

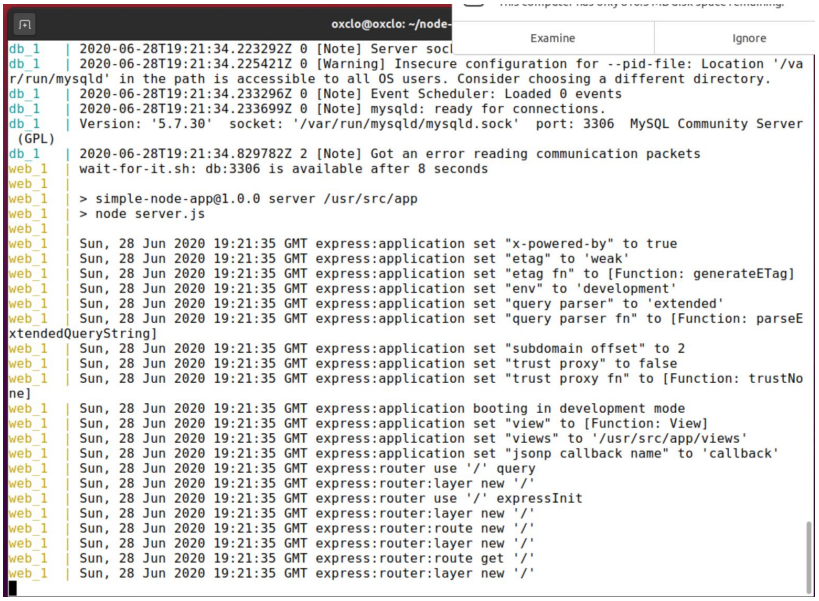


I won't explain everything, but here are some key points:

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

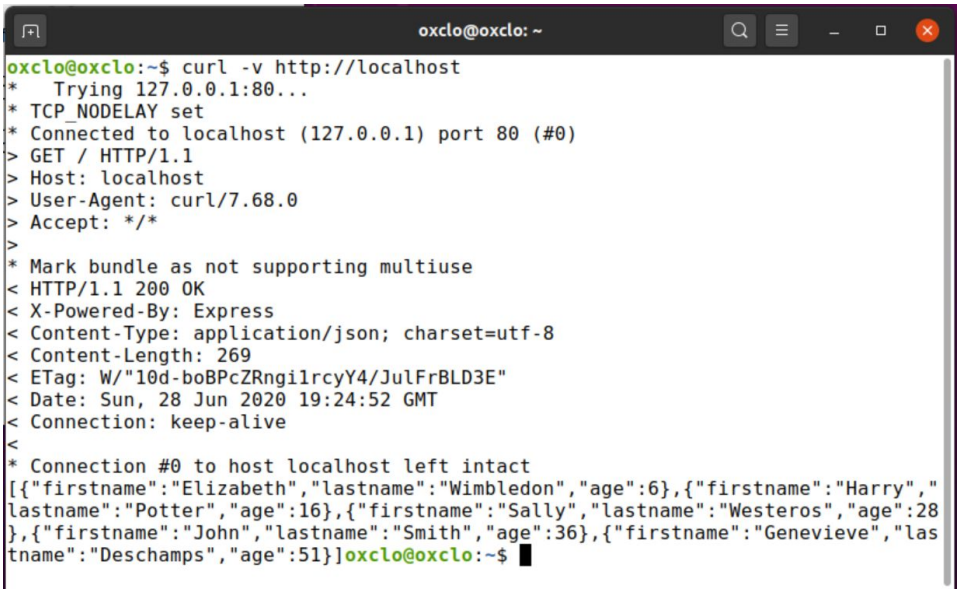
52. Now let's start the "composition"
    `docker-compose up --build`

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



54. In another window try:
    `curl -v http://localhost`

    You should see successful query of the database



    You can clear up by typing:
    `docker-compose down`

Congratulations, you have completed part B.

PART C – Running your docker in the cloud

55. There is an Amazon Elastic Container Service for running containers. This provides their own container orchestration model, alternative to Docker Swarm and Kubernetes. However, we are going to use Docker Machine and Docker to start our systems instead, giving us a portable approach.

56. If you already did Exercise 1, then your AWS credentials should be stored in ~/.aws/credentials. Docker Machine uses those so you don't need to set any environment variables.

57. Type:

```
docker-machine create \
   -d amazonec2 \
   --amazonec2-region eu-west-1 \
   --amazonec2-security-group web-server-sg \
   oxcloXX-docker
```

This ensures our server will run in Ireland, and use our security group.The instance will be called oxcloXX-docker.

You should see:

```
Running pre-create checks…
Creating machine…
(oxclo01-docker) Launching instance…
Waiting for machine to be running, this may take a
     few minutes…
Detecting operating system of created instance…
Waiting for SSH to be available…
Detecting the provisioner…
Provisioning with ubuntu(systemd)...
Installing Docker…
Copying certs to the local machine directory…
Copying certs to the remote machine…
Setting Docker configuration on the remote daemon…
Checking connection to Docker…
Docker is up and running!
To see how to connect your Docker Client to the
     Docker Engine running on this virtual machine,
     run: docker-machine env oxclo01-docker
```

58. Go to the AWS Console and you should see an EC2 instance now running.

59. Follow that last suggested command:

```
docker-machine env oxcloXX-docker
```


You should see:

```
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://54.246.240.90:2376"
export
DOCKER_CERT_PATH="/home/oxclo/.docker/machine/machines/oxclo0
1-docker"
export DOCKER_MACHINE_NAME="oxclo01-docker"
# Run this command to configure your shell:
# eval $(docker-machine env oxclo01-docker)
```

60. Once again, do as asked:
    `eval $(docker-machine env oxcloXX-docker)`

61. Now when you run docker commands, they will no longer act locally on your Ubuntu VM, but instead on the AWS instance you've just started.

62. Now run the docker container again:
    `docker-compose up --build`

    While it builds you can think about two things:
    - Why is it pulling all those image layers when you've already done that when you ran docker-compose last time?
    - Why is it pulling them much faster than before?

      The answer to both questions is the same: this time docker is running on the AWS instance (which probably has better internet bandwidth than you do).

63. You can find the address of the AWS instance using:
    `docker-machine ip oxcloXX-docker`

    or by looking in the AWS console

64. Open a browser window and check you can access the app (running on port 80)

    Alternatively this command will open a browser from the command line pointing at your instance:
    `xdg-open http://$(docker-machine ip oxcloXX-docker)`

65. You can also SSH into the server using:
    `docker-machine ssh oxcloXX-docker`

66. Try `docker ps` from there

67. Try that. When you are finished type
    `logout`

68. To terminate the instance, type:
    `docker-machine rm -f oxcloXX-docker`

69. Check that the instance is now terminating in the AWS console.

70. Docker Machine will have automatically created a new key pair for this server (which is not ideal….). To tidy up, lets remove that:

    ```
    aws ec2 delete-key-pair --key-name oxcloXX-docker
    ```

71. We have now seen how we can containerise not just single processes but complex networked apps using multiple components. In the lectures we will address how to cluster and run these across multiple servers (using "cloud orchestration" and Kubernetes.

72. What we have done is to take the container-ised application we built on our local machine and then deploy it automatically to EC2. While this looks a bit similar to using userdata (which we did in a previous exercise), in fact this is much more replicable because we can test and deploy our docker image on many different systems. For example, if you sign up to the Github Student deal, you get free credit on DigitalOcean, and with a very similar model you could run the same docker code on there.

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.