**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

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



1. Notice how little memory each container takes. This means you can run hundreds of containers on a normal machine.
2. Now **Ctrl-C** to exit that command.
3. 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.
4. Once you have signed up, then do a docker login:  
   docker login -u *yourdockerhubuserid*
5. 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
6. Now list the images you have locally  
     
   docker images
7. 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
8. Actually it would be useful to give that image a version name:  
     
   docker tag yourdocker/mycentos yourdocker/mycentos:1
9. Repeat the “docker images” command.
10. Now let’s push that image up to the docker hub:

docker push yourdocker/mycentos:1  
  
*Enter your docker hub credentials if prompted.*

1. The system will whirr away and upload some stuff. Eventually you will see something like:  
   
2. Now let’s go back to the original docker window, where your image is still running. Make a new file in home like this:  
   
3. Now in your control terminal you can commit this change:  
     
   docker commit cranky\_beaver yourdocker/mycentos:2
4. Let’s push that image you’ve just made up to the Docker hub:  
     
   docker push yourdocker/mycentos:2
5. 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.
6. Go to the docker website <http://hub.docker.com> and view your repositories. In particular look at the tags tab:  
   
7. 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.
8. 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

1. Exit that container as well.
2. 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/\*)
3. 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.
4. The one thing we haven’t yet seen is how to get a docker image to do something vaguely useful.
5. First check you have nothing running locally on port 80. Browse to <http://localhost:80> It should fail.
6. Now in your docker window, type:  
     
   docker run -p 80:80 httpd
7. You should see a bunch of stuff like this:  
   
8. Now browse <http://localhost:80> again and you should see.
9. *Are you wondering what –p 80:80 means?*It means expose port 80 from within the container as port 80 in the host system.
10. 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
11. Test <http://localhost> again
12. 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



**PART B – Building a container using a Dockerfile**

1. 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.
2. Clone the git repository:  
   git clone https://github.com/pzfreo/node-docker.git

1. Then

cd node-docker

1. 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

1. Now  
   docker build -t *<your\_docker\_id>*/nodeapp:1 .

(notice the ‘ . ’!)

1. While it is building, take a look at the docker file and also the reference guide:  
   <https://docs.docker.com/engine/reference/builder/>
2. Once it has built, try running it:  
     
   docker run --name nodeapp -d -p 80:8080 <yrdockerid>/nodeapp:1
3. 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"}

1. Kill that container:  
    docker rm --force nodeapp
2. 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.
3. So to make our app work, we need to run two containers:
   1. node and
   2. mysql
4. 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.
5. Look at docker-compose.yaml  
   code docker-compose.yaml  
     
   

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

* 1. We have two “services” which will be container runtimes: “web” and “db”
  2. Web depends on db, so won’t start until the other container is started
  3. 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.
  4. We configure everything through environment variables, especially the links between the containers
  5. 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.
  6. 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

1. Now let’s start the “composition”  
   docker-compose up --build
2. You should see lots of logging go by until finally you see:  
     
   
3. 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.