Docker Worksheet

Alex Wright & Matthew Mostert
FEEG6003

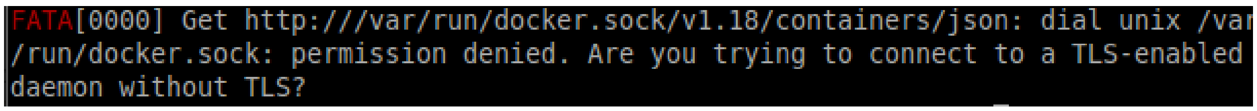
May 6, 2017

Try the following exercises as instructed. To complete the exercises, you will need to use the virtual machine provided at http://www.southampton.ac.uk/ ngcmbits/virtualmachines/feeg6003Docker.ova . Enter the following:

```
$ docker
```

to see if the VM is working correctly. You should see a list of commands if docker is ready.

If at any point throughout this worksheet you run in to the following error,



```
FATA[0000] Get http:///var/run/docker.sock/v1.18/containers/json: dial unix /var
/run/docker.sock: permission denied. Are you trying to connect to a TLS-enabled
daemon without TLS?
```

prepend the command with

```
$ sudo [original command]
```

or, alternatively, execute

```
$ sudo -s
```

before continuing (recommended).

Exercise 1. We begin with the obligatory hello-world example. Tutum provide such an image so we can see how to build a simple container. First, we need to pull the image from Docker Hub. We could do this using the *pull* command, but if a repository cannot be found locally, Docker will automatically search and pull the repo from the Docker Hub.

```
$ docker run tutum/hello-world
```

To check this container is running, open up a new terminal window and use

```
$ docker ps
```

This command displays all active containers, and we can see that the output of the container is coming from port *80*. The flag *-a* will also display any inactive containers. At the moment, the container is running in the foreground. We can make containers run in the background with the flag *-d*, where *d* stands for daemon[1].

In order to see the output from the daemon, we need to re-run the hello-world image exposing the ports we want. Use the *stop* command with either the container ID or name to kill it. Re-run the image, but now run it in the background and expose port 8080[2].

We can see if this is working by going on to the browser and viewing the output from the local host. Into the address bar, type *localhost:8080* to see if the output is being displayed correctly.

Stop this container and refresh the browser - why is has this changed?

Docker containers can be run multiple times - this is one reason why Docker is becoming very popular with devs, as anything produced with docker is completely scalable. Re-start the previous container, and set up another hello-world container, but this time set the output to 8081. What is the difference between these two outputs?

Kill and remove both these containers - check that all containers (including inactive ones) have been removed.

Exercise 2. All science relies on reproducibility, and this is no different in computational science. It is essential that any results generated via computational methods are able to be reproduced by others on their own machines. This is where *dependancy hell* becomes a nuisance [1]. Docker is a very useful tool that we can employ during scientific programming that makes it far easier for others to use our work, and verify our results.

For this example, we shall use a scientific python package from the group at www.dataquest.io. First, pull the python3 layers from Docker Hub with

```
$ docker pull dataquestio/python3-starter
```

Now, create a new folder in which to save any work or output (for example, *path/output*). We can now run this image and begin work on our document. Run the image we have just pulled as a daemon, re-route the port 8888 to 8888 so that we can see the output, and use the *-v* flag to re-route where the documents are to be saved locally[3]. Check this works by using your browser (as in exercise 1).

Now, you should be able to create a new notebook - call it *path* and insert the following python code:

```
import os
print(os.getcwd())
```

What do you get when you run this? Save the notebook and check that is has been saved to the directory you specified. Why do these locations differ?

Stop the container and check this with *docker ps -a*. Refresh the browser to check you get what's expected. Now make a change to the file on your local machine. Do this by going

[1]Your guess is as good as mine

[2]Hint: the *-p* flag tells docker to re-route the output, specify the output by following the flag with *desiredOutput:currentInput*

[3]Hint: Use the command *-v path/to/desired/directory:standard/output/directory*, where the standard place for documents to be saved is */home/ds/notebooks*

to the directory in which you saved the notebook and opening up the *.ipynb* file in a text editor. In the *source* block, where you were printing out the current working directory with *os.getcwd()*, replace this with a string (for eg. some Oasis lyrics) and save.

Changes to these files may be done whist the containers are active or inactive - it is not essential to run your programmes on containers all the time, but errors will be noticed sooner and fixed earlier if you do so. For example, if importing data in to a file locally, this will not automatically work in the container. But if it works in the container, it will work automatically on your local machine.

What do you get now when you start the container and refresh the browser?[4]

Scientists rely on data, and if your research uses some set of data to analyse this should accompany your code. There are multiple ways of including data sets in your container:

1. Copy the data file into the location that you are saving any output.
2. Use the *docker cp path/to/data/file containerID:/home/ds/notebooks* command. You can get the container ID from the *docker ps* command.
3. Use the upload tab on the top right of the Jupyter home window.

Use all three of these methods to copy the data files from the */Data/* folder for use in the notebook. Ensure that you can see these files on the Jupyter home screen.

Exercise 3. So far we have been using open sourced images for our examples. These Docker images have been prepared and built by various devs, uploaded onto a remote server (often Docker Hub), which we then download and run on our machine. To make our research available in a similar manner we will need to be able to build our own images.

Building images starts with a Dockerfile. These are purely text files with the name *Dockerfile* (no extension) that contain all the files, data and commands required to build an image. With the inclusion of some basic commands specific to Docker, including FROM, RUN, and CMD, the Dockerfile is essentially a bash script. A sample script is available on the VM provided. Have a look through the programme $simple-prog.py$ using a text editor.

The VM provided can run standard python scripts, but does not have available to it scientific packages that we may use for our research. This would be a common problem among researchers that use very specific software in their research. Also, provided is a sample python script that uses the NumPy module. Try to run this script in the normal way.

Also provided is a Dockerfile. Use an editor to open the file and understand each step within it. Images are built from such text files using the *dockerbuild* command, one must specify the location of the Dockerfile, and any additional tags in the command. Build the image with the relevant command, and use the tag $-t$ *sample.image* to label the resulting image. Check this has worked with *docker images*.

Images can be made public by pushing them to a repository on Docker Hub. (See live example)

[4]Careful: If you get an *Unreadable Notebook* error, you may have forgotten to put backslashes before the double quotations.

The $simple - prog.py$ script is contained in a docker image, accompanied with the numerical python library, and been made available on Docker Hub. Spin up this container using the

```
$ docker run REPOSITORY[:TAG]
```

command, baring in mind that the repo has the address *ajw1e16/docker.workshop* and the image has the tag *sample.image*. The image created from the Dockerfile provided has the numerical python library installed, so when the container is executed, the programme does not return any errors.

[1] https://arxiv.org/pdf/1410.0846.pdf