CAB432 Cloud Computing

Week 2 & 3: Getting Started with Docker

Docker

Docker is an open-source tool that automates the deployment of applications inside software containers. It adds an extra layer of abstraction and automates OS system level virtualisation on Linux. The beauty of Docker is that it allows you to deploy a prebuilt image very quickly – it takes care of installation of all necessary packages and starts running the application. This allows you to provision new servers in a cloud scenario without a lengthy configuration burden.

Docker now comes in a couple of different guises – one targeting enterprise customers (paid) and one more open and labelled as the community edition (free), which is the one we will use in this unit. In the exercises below we will work with Docker under Linux and we will consider the various ways you can access a Linux environment. But you may also run Docker under (recent versions of) Windows or Mac OS. Please see the instructions and links in the appendix for details of these alternatives. We won't directly support them and the prac below assumes that and you will need to use a specific Linux VM for Assignment 1, but you probably won't have too many issues. Some screenshots may have minor inconsistencies due to command updates (Python 3).

Exercise 1 – Getting Started

The basics of using Docker are covered very nicely in Docker's own tutorials. The full tutorial will take about 30 minutes if things go according to plan.

In this unit, therefore, we will work mainly with the Linux version of Docker, and the supported version will be Ubuntu LTS 18.04. Most recent versions of Ubuntu (including 20.04) will work painlessly here. It is a requirement of this prac exercise, and of course assignment 1, that you are able to deploy your Docker container to the public cloud. So, at some point you will need to fire up a VM on AWS or one of the other public clouds. Preparation of the Docker image, however, can take place anywhere you can access a tame Linux workstation. There are four main options:

- Install Linux natively on one of your personal machines.

- Use Docker under the Windows Subsystem for Linux (see: https://code.visualstudio.com/blogs/2020/03/02/docker-in-wsl2 and note the comments in the appendix that this doesn't work well on the QUT lab machines).
- Install Linux as a guest on your Windows machine using Virtual Box https://www.virtualbox.org/ or an alternative VM manager. We provide some basic guidance on this in the appendix.
- Use an additional, separate Linux VM to prepare the image as well as firing up a VM for deployment of the container after we have pushed it to Docker Hub. This is the approach we will use in this guide, and what you should do if you don't have a laptop with you in the labs.

There are two pre-requisite tasks:

- Installation of Docker – this is essentially the same as the approach in Guide to working with the QUT managed AWS machines, though you can use the `apt-get` mechanism if you prefer.
- Creating an account on Docker Hub – similar to GitHub for storage of your Docker images; once they are pushed to Docker Hub they can be installed and run anywhere e.g. in a cloud server instance. This is now wrapped into the Docker Cloud but the Hub is still the service we require.

More information on all aspects of using Docker is available at the docs pages:

https://docs.docker.com/.

The getting started guide at https://docs.docker.com/get-started/ is particularly good on the terminology and concepts. We have discussed these in the lecture, but it is good to remember the distinction between the container and the image. You may also see, in various places, mentions of Docker services for the cloud platforms AWS and Azure. Please ***do not*** use these. They are set up for orchestrated deployment at scale, integrated with the cloud services. They are important but they are not suitable as a starting point. Let's understand the basics first.

Installation

Here we will follow the installation process for Docker we used in the AWS Guide. As before, there may be some differences between the screenshots below and what you will see in the QUT environment.

Docker is fundamentally a system built for use under Linux, and you should experience few problems installing under a native Linux environment. The relevant section from the AWS Guide is included below. This should also work painlessly from local installations of Ubuntu.

We will begin by installing the application Docker, and running a basic sanity check deployment of `hello-world`*. Docker exists at: https://www.docker.com/. To install new software, we*

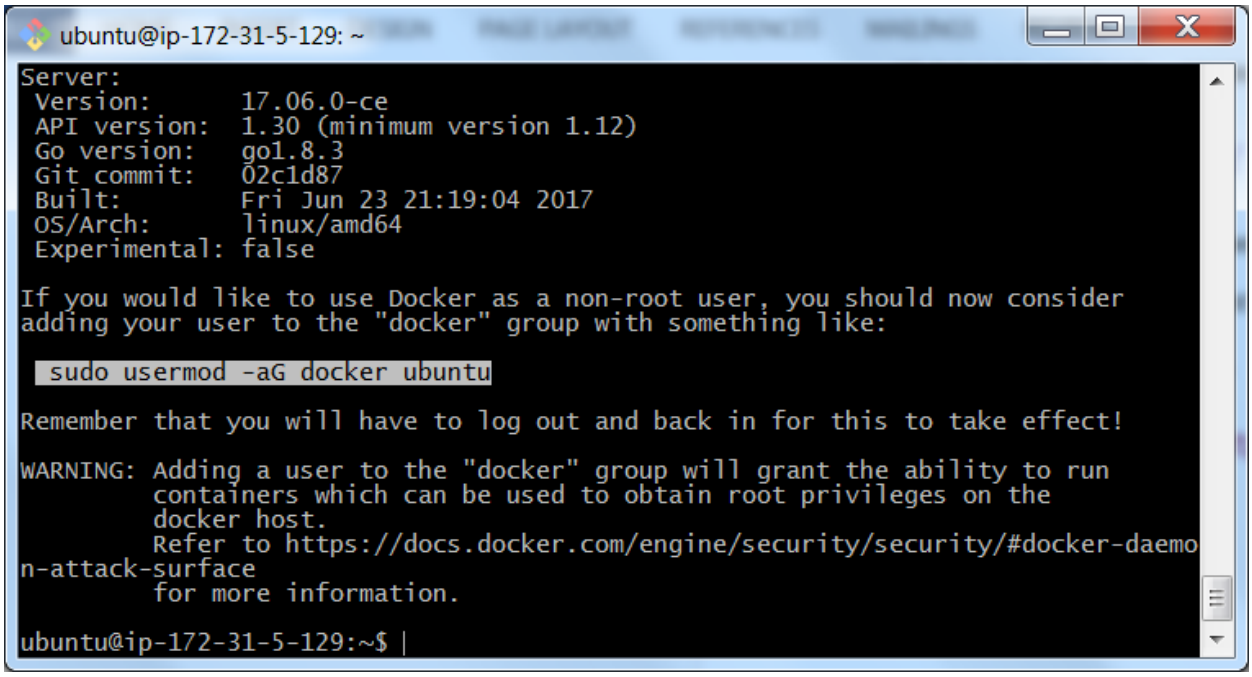
would normally use the `apt-get` application update system, but here there are more convenient alternatives – don't worry about the commands at this point if you don't already know them.

```
sudo curl -fsSL https://get.docker.com/ | sh
```

Your machine should have curl installed by default so it shouldn't be necessary to check. We use the `sudo` command to ensure that the installation scripts have elevated privileges. Once the Docker installation is complete, you should run a basic Docker hello world to confirm that everything is running as expected.

During the installation process, you may be offered the chance to add a username to the Docker group. If you are able to do this, make the assignment for your chosen username. If you did not do this, then you will need to run the Docker commands below prefixed by `sudo`. I will not include this prefix in any of the commands that follow, but do be aware that it may be necessary. If you encounter the error that Docker cannot connect to the Docker daemon, this is almost certainly the issue, and the use of `sudo` (though not exactly ideal practice) will allow you to proceed.

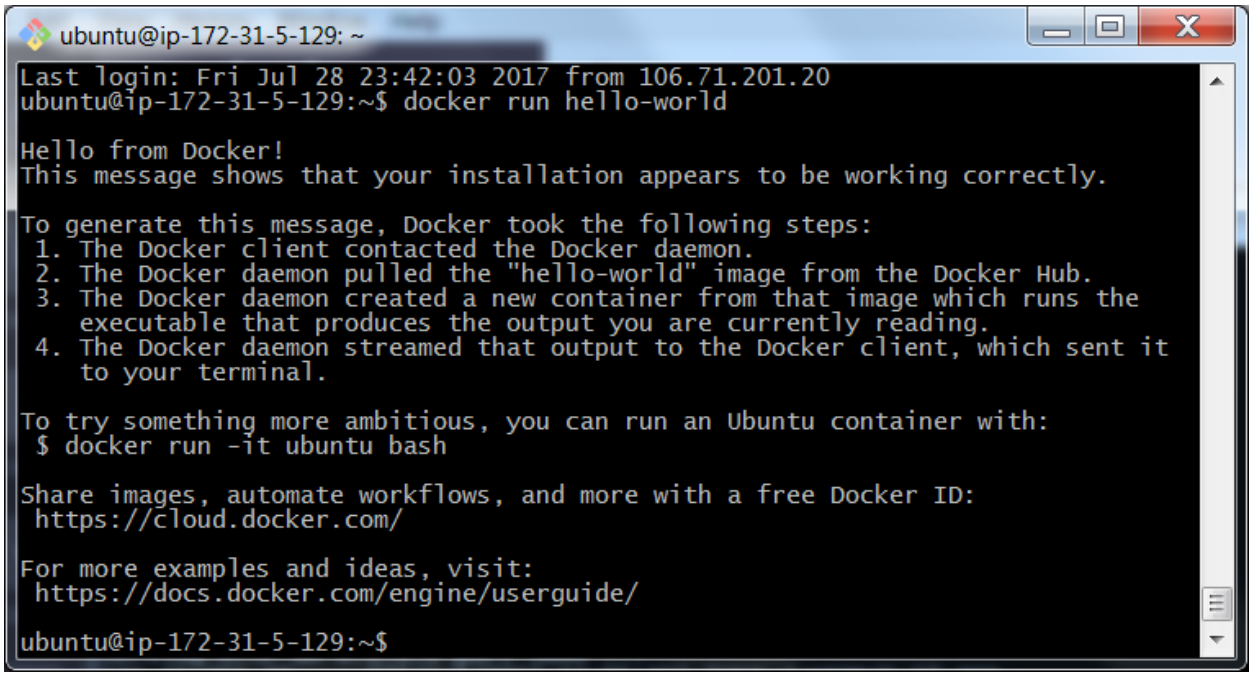
After successful installation of Docker, you should see something like this:



Here I have highlighted the instructions to add the main user `ubuntu` to the Docker group. This executes painlessly, and we are then able to go ahead and run the hello-world example without using `sudo`, *but only after logging out and logging back in*. The command is once again simply:

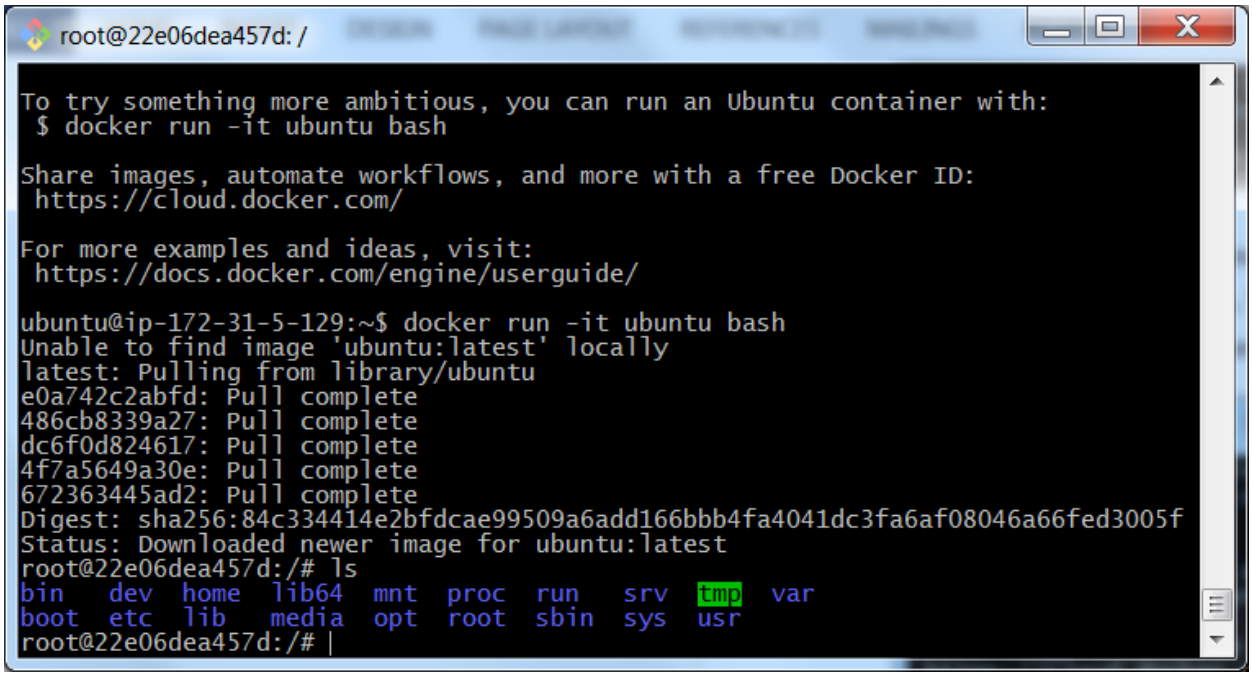
`docker run hello-world`

and if all is going according to plan, we will see something like this, with an additional message if the image was not available locally and had to be pulled down from the web.



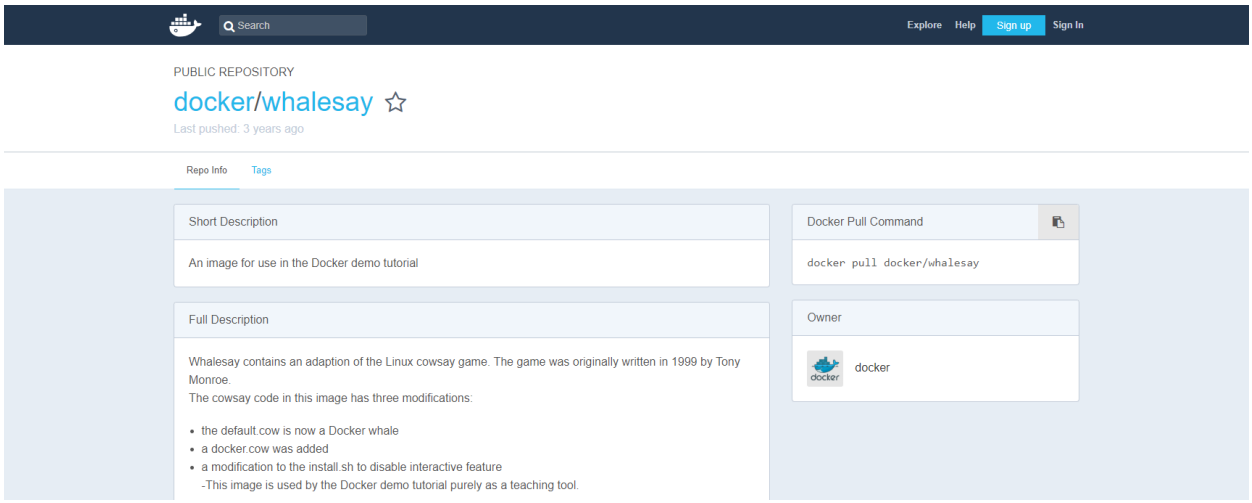
We may then go ahead and try the Ubuntu image. The Ubuntu image is the first of the serious images we will work with. We don't initially have a local copy and so it grabs the image from a remote store at the Docker Hub.

`docker run -it ubuntu bash`



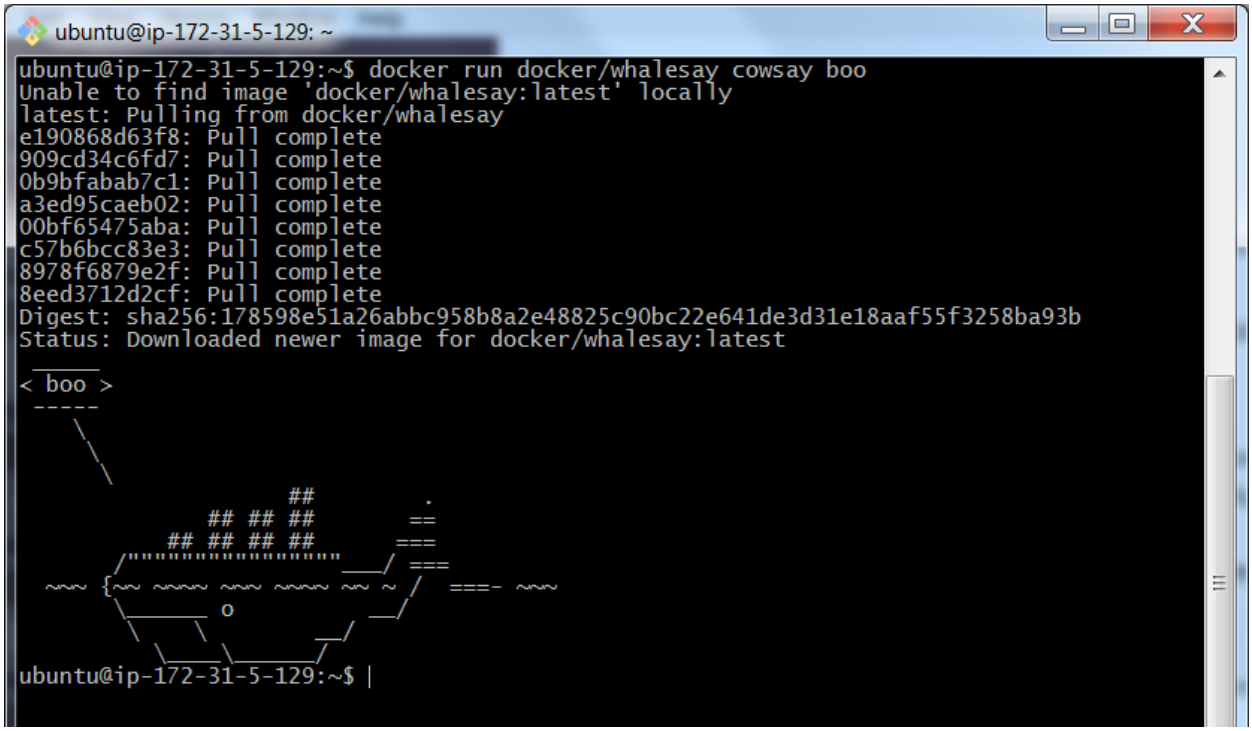
Note the change in the prompt from the `ubuntu@ip` form to the `root@22e...` form shown at the bottom of the screen. We are now running Ubuntu from the Docker container sitting on top of Ubuntu on the AWS VM. Finally, we continue through the original Docker tutorial and use the Whalesay image, updating it as required to use the fortunes application, an approach which will then allow new sayings to appear each time we invoke the image.

We begin by working with the vanilla Whalesay image. The application is trivial, and is an adaptation of an earlier unix sayings app called cowsay. The details of Whalesay can be found at https://hub.docker.com/r/docker/whalesay/:

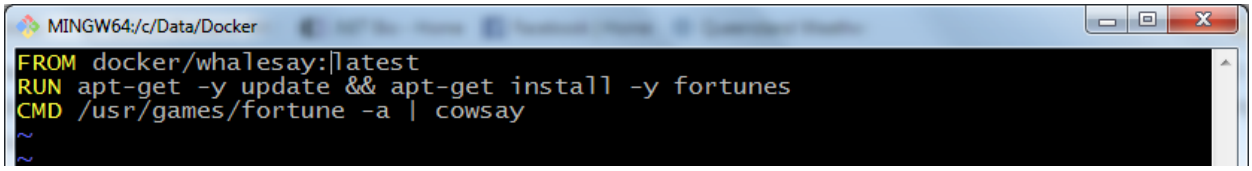


Don't take it seriously. Execute the command and we see the screenshot below it:

`docker run docker/whalesay cowsay boo`



We are now going to create a simple `Dockerfile` following the conventions introduced in the lecture. We will grab the latest version of `whalesay` – nothing much will happen in our case as we will already have it locally – and we will then grab a simple game called fortunes, which unsurprisingly produces quotes randomly from a database.



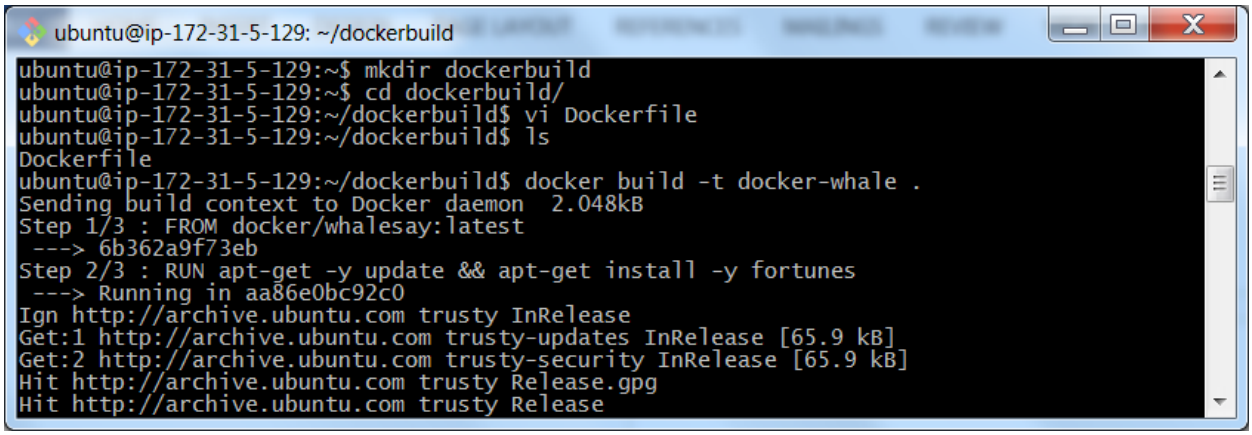
Note the syntax in the `RUN` command. (Also the screenshot is confusing – the mouse cursor is near `latest` – there is no pipe character after the colon). We are first making sure that the system is up to date and only then installing the application that we want. Note that this will be a common theme in Docker related exercises. Precise instructions on creating the `Dockerfile` and then using Docker to transform this into an image are found here (you may have to scroll up a few lines):

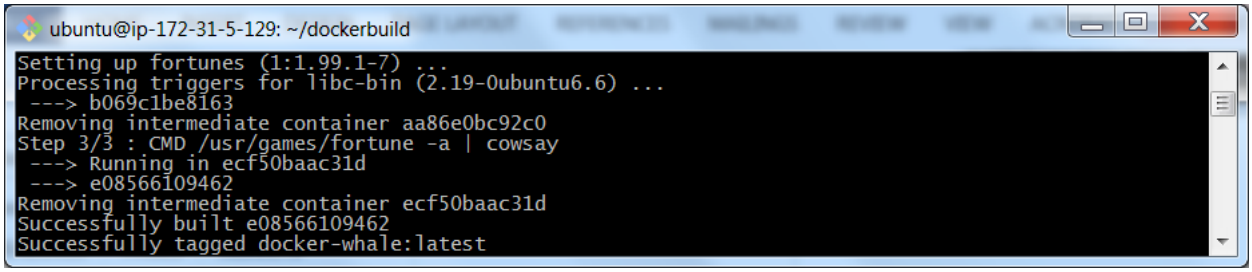
https://medium.com/@deepakshakya/beginners-guide-to-use-docker-build-run-push-and-pull-4a132c094d75#6562

In general, you will need to build your image with ***`docker build -t docker-whale .`*** (here we've tagged our image "docker-whale").

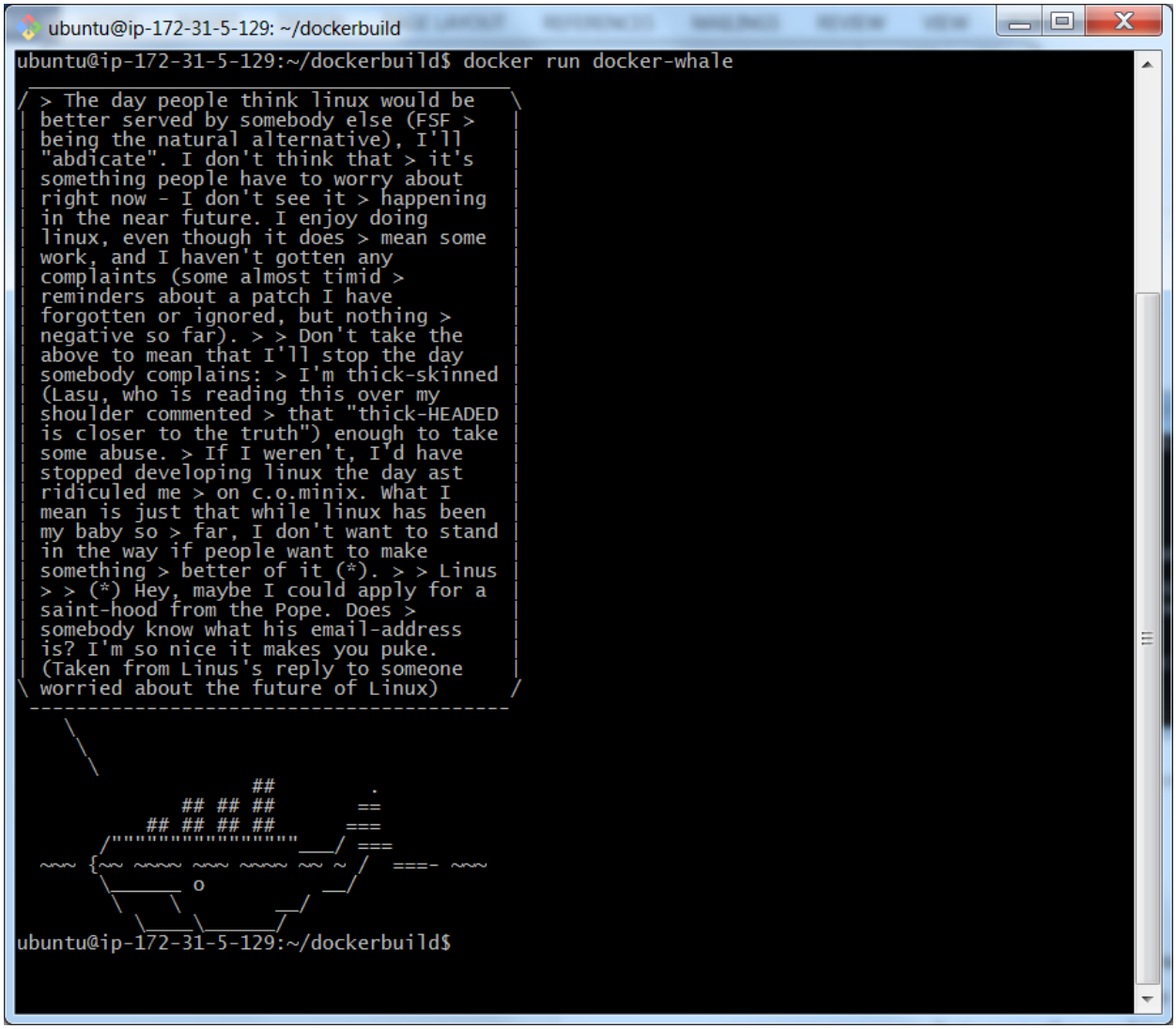
You will at some point need to come to grips with a Linux-based editor. Editors are a matter of deep loyalty, as once you have learnt one very well you become extremely reluctant to change. In my case, I use vi as I learnt it shortly after birth, but others may use nano or other alternatives. See the appendix for some links to tutorials, and feel free to suggest others.

My session is shown below. The output is dominated by the update, some of which I have not shown. Pay particular attention to the steps corresponding to the commands in the `Dockerfile`, and the intermediate hashes that appear at each stage. This means that if there is an error later on, we don't have to rebuild everything from scratch when we start again.





At this point, we can go ahead and run the new image, which in this case produced probably the longest ever fortune entry, a quote from Linus Torvalds:



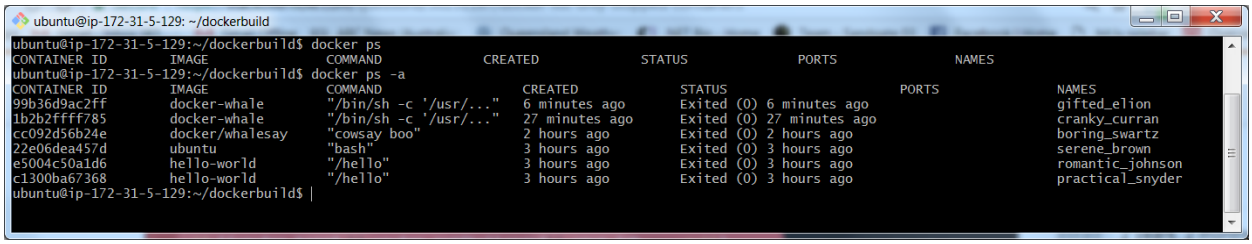
Finally, the tutorial tells you to create your own Docker Hub account (yes) and then to push your own copy of this new version of the whalesay app (no). Please do go ahead and create your own account (see https://medium.com/@deepakshakya/beginners-guide-to-use-docker-build-run-push-and-pull-4a132c094d75#44f4 – you may again need to scroll up a few lines) but save the pushing for stuff that matters, the images we create in exercises 2, 3 and 4.

Before moving on to the more complex matters, it will probably help to explore some of the more useful Docker commands and perhaps to clean up a little. There is a reason for this, as we

will see shortly. Mostly Docker follows very standard Linux command line conventions, and the main commands are readily seen using the conventional

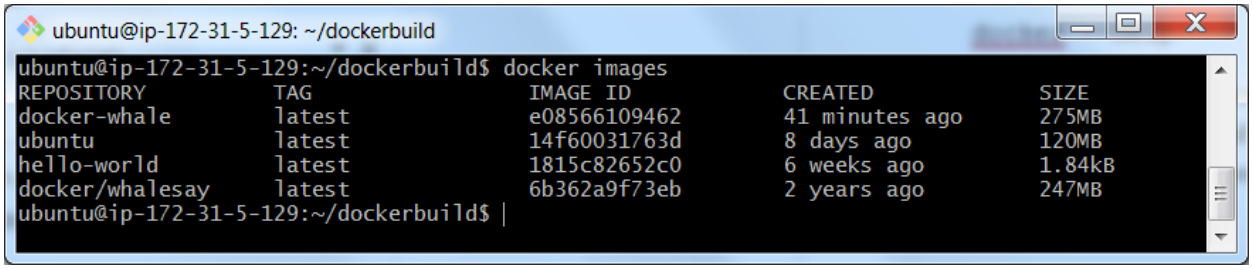
`docker --help`

This generates a substantial list of commands and alternatives. We will focus on those that manage the images and the containers. To follow through on these exercises, login to the same VM using a second terminal window, or create another xterm if you are using the machine locally. This will allow us to monitor the machine when we have a running Docker container. At present we have no containers running (***`docker ps`***) but by using ***`docker ps -a`***, we can see all of the containers that I have used over the course of the session:

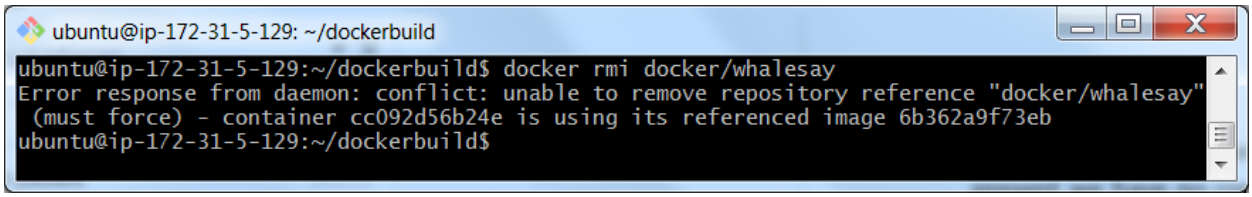


Note the new equivalent version of this command: ***`docker container ls`***

Take note of the hash IDs for some of these containers. In a moment I am going to look at some images, and try to delete the one for `docker/whalesay`. Here of course we see that this image is associated with the exited container `boring_swartz` (container names are obviously enough, auto-generated), which has ID `cc092d56b24e`. Let us now list the images.



I don't really care at all about the whalesay image and so I am going to destroy it. The ***`docker rm`*** command removes containers; here I want to remove images, and the command is similar, ***`docker rmi`***. We go ahead, but run into trouble:



This time, as I am sure of the consequences, I will force:

```
ubuntu@ip-172-31-5-129: ~/dockerbuild
ubuntu@ip-172-31-5-129:~/dockerbuild$ docker rmi -f docker/whalesay
Untagged: docker/whalesay:latest
Untagged: docker/whalesay@sha256:178598e51a26abbc958b8a2e48825c90bc22e641de3d31e18aaf55f3258b
a93b
ubuntu@ip-172-31-5-129:~/dockerbuild$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker-whale        latest              e08566109462        44 minutes ago      275MB
ubuntu              latest              14f60031763d        8 days ago          120MB
hello-world         latest              1815c82652c0        6 weeks ago         1.84kB
ubuntu@ip-172-31-5-129:~/dockerbuild$
```

We now look at `docker-whale`, and run it again using the usual command:

```
ubuntu@ip-172-31-5-129: ~
ubuntu@ip-172-31-5-129:~$ docker run docker-whale
 ______________________________________
/ In any world menu, Canada must be    \
| considered the vichyssoise of nations |
| -- it's cold, half-French, and       |
| difficult to stir.                   |
|                                      |
\ -- Stuart Keate                      /
 --------------------------------------
    \
     \
                    ##        .
              ## ## ##       ==
           ## ## ## ##      ===
       /""""""""""""""""___/ ===
  ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
       \______ o          __/
        \    \        __/
          \____\______/
ubuntu@ip-172-31-5-129:~$
```

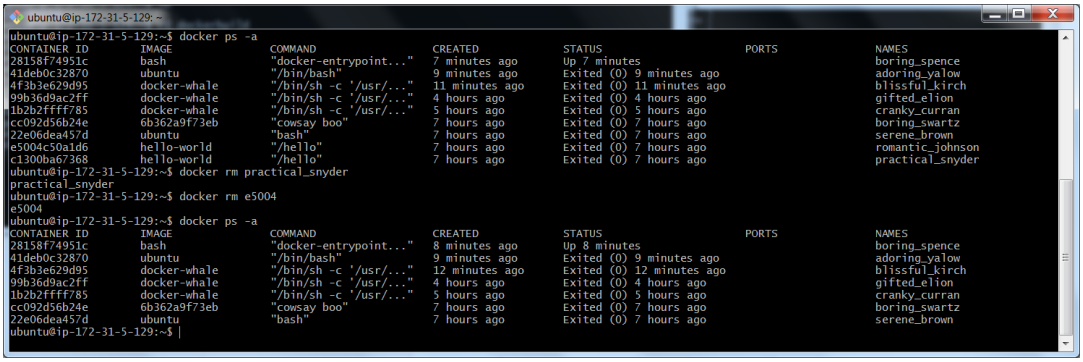
We see that it works just fine – the fact that `docker-whale` builds on the earlier `docker/whalesay` image is not an issue. The new Docker image is independent of the earlier one. We now run the Ubuntu image and monitor it from the other login terminal. Here the bare ***`docker ps`*** command shows a running container.

```
ubuntu@ip-172-31-5-129: ~
ubuntu@ip-172-31-5-129:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
28158f74951c        bash                "docker-entrypoint..."   56 seconds ago      Up 56 seconds                           boring_spence
ubuntu@ip-172-31-5-129:~$
```

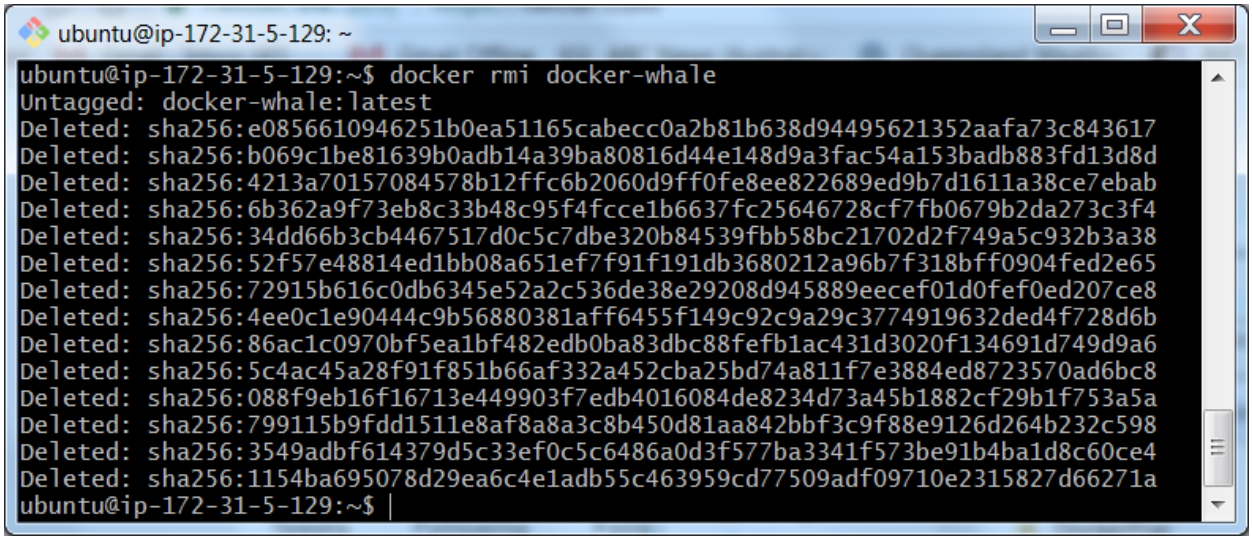
An attempt to remove the container meets with some issues:

```
ubuntu@ip-172-31-5-129: ~
ubuntu@ip-172-31-5-129:~$ docker rm boring_spence
Error response from daemon: You cannot remove a running container 28158f74951c912b3e10468db80af5290c8c00d1f1b86120eb73e49749a0ff14. Stop the
 container before attempting removal or force remove
ubuntu@ip-172-31-5-129:~$
```

As before, we can force, but we will choose not to. If we want to, we can remove the others. Here I remove `practical_snyder` using the tag name, and `romantic_johnson` using a unique prefix of the hash ID:

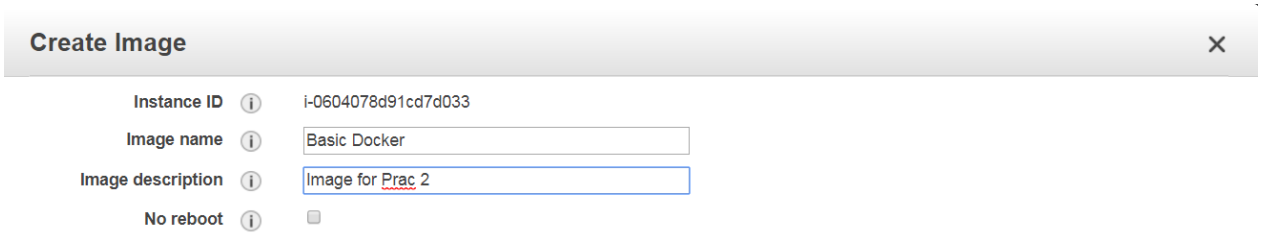


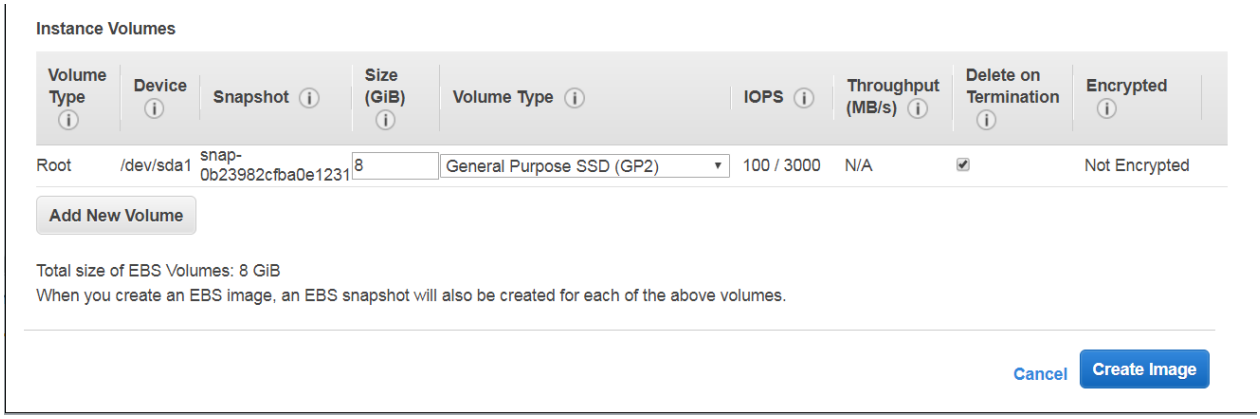
I then proceeded to clobber all of the containers other than the one presently running. Following on from this, we can revisit the idea of deleting an image. I don't want even the modified `docker-whale` image, and so, this one is my main target:



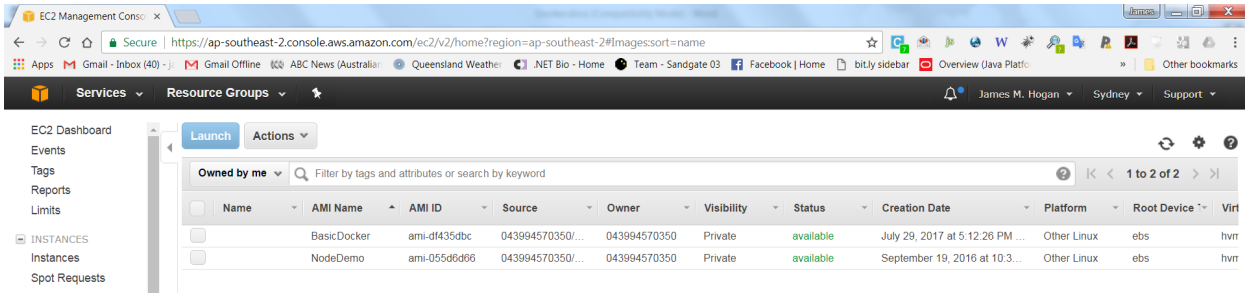
Note that killing an image will also remove the hashes corresponding to the states encountered in its creation. At this point, we still have a nice running Ubuntu server with Docker installed and an Ubuntu image available. Over the coming week or two we may find ourselves wanting to reuse configurations. Sometimes this is best done at the Docker container level, sometimes at the level of the VM itself. Here we will learn how to preserve a VM.

The process is simple. Go to the EC2 instance view and use the right click menu on your instance, selecting *Image > Create Image*, leading to the Create Image dialog. Here I give the image a name (revised later to exclude the space), a description and we preserve the basic disk structure.



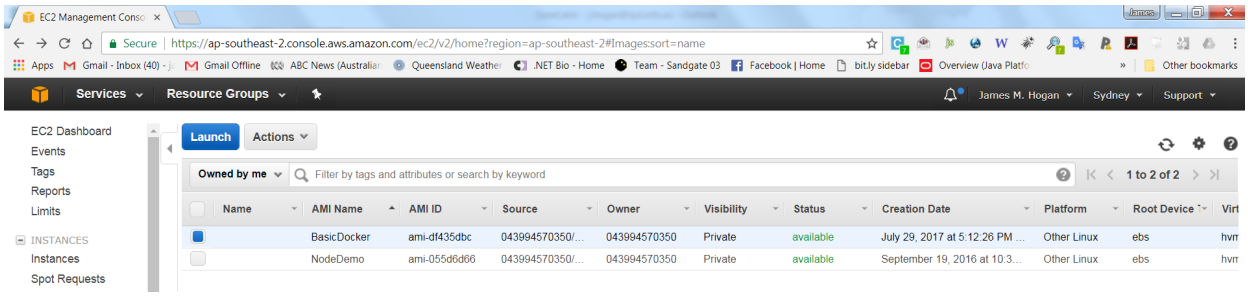


Once the image is created, we go to the menus on the left hand side and select AMIs under the Images heading. In my case we see two images, one saved from last year. The cost of storing these images is very low: https://aws.amazon.com/ebs/pricing/.



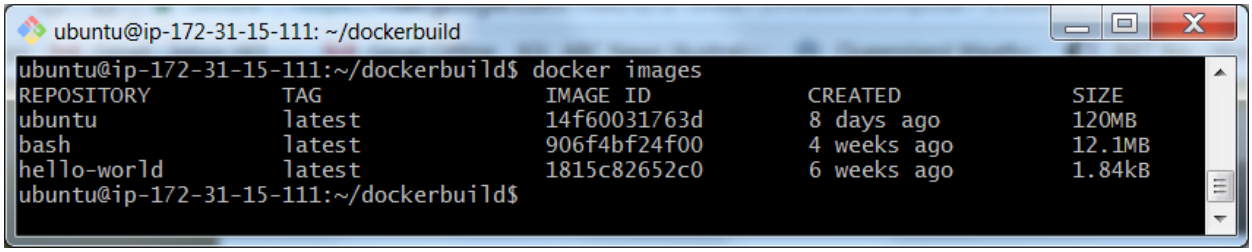
Suppose now we want to resurrect this image and use it for later work? In my case, I finished this section earlier this afternoon and have now come home again, and so I can fire up the server. The key in using such a machine – and keeping costs down while doing so – is to not maintain the active storage and to not preserve the IP address. We treat it just like another machine image like those in the menus, with IP and other settings to be added during launch.

At this stage, we select the image and enable the launch button. We click to follow pretty much the standard process. Make sure that you have a suitable custom TCP rule in place in the security settings to allow http access to the machine. Here you should open port 8000. You will need this to verify that the exercises are completed correctly. See the port mapping later in the prac.



After launching and following the usual login procedure – note that AWS may use `root` as the default username in the connect example and you will have to change to `ubuntu` – we find the

image as expected. Here I have changed into the `dockerbuild` directory and run the `docker images` command to show that the earlier images are preserved:



We are now ready to start work on the more sophisticated exercises which make up the rest of this prac. Once again we will build using `Dockerfiles` rather than using `docker compose`. Both approaches have their merits, but for now I think that the direct use of the `Dockerfile` is a better teaching approach. We will now build a Python server – this will be much more sophisticated than the very simple approach you took in the first prac. But it is still well short of a real web server usable in a production environment.

Exercise 2 – Ubuntu with Python Server (Hello World)

In this exercise you will build a Docker image based on Ubuntu Linux, install Python, and create a simple Hello World Python app which gets served by a Python Web Server Gateway Interface (WSGI). You will be able to view the output in a browser by navigating to the IP address of the underlying Docker machine (see below).

You will find the following link very useful in understanding `Dockerfile` creation and the conventions to follow: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/. In particular, this provides really good background on the use of `apt-get` and the structures to use to avoid caching issues and `Dockerfile` creation failures. I strongly recommend that you follow the recommended conventions with respect to the lexically ordered, one app per line construction of the `apt-get` arguments.

Let us begin as follows:

1. Fire up the AMI you saved in the previous exercise or use your local Linux machine or VM.
2. Return to the Docker terminal command prompt.
3. Type ***`docker images`*** at the command prompt to view your current list of images.
4. Navigate to your Docker work area and create a directory named `exercise2`. This directory will contain the `Dockerfile` which creates the image and the necessary files to run the Python application.
5. The following directory structure is required. It will sit alongside `Dockerfile`.

```
/app
    |- /app
    |- app.py
    |- server.py
```

6. `app.py` and `server.py` have been provided on Blackboard. Copy them into the `exercise2/app` directory.

7. Create `Dockerfile` in the `exercise2` directory and open for editing.

    a. We are building an Ubuntu image. The `FROM` command is used for this

```
##############################################################
# Dockerfile to build Python WSGI Application Containers
# Based on Ubuntu
##############################################################

# Set the base image to Ubuntu
FROM ubuntu
```

    b. The file author should be included as with all software.

```
# File Author / Maintainer
MAINTAINER Your Name
```

    c. Following the guidelines in the best practices doc, we now create an extensive apt-get task which performs an update to Ubuntu and grabs specific applications, with each of these laid out in alphabetical order to allow easy updates and maintenance. Pip is a package manager for Python. We get it to install the modules needed for this exercise.

```
# Install basic applications, Python, Python tools
RUN apt-get update && apt-get install -y \
   build-essential \
   curl \
   dialog \
   git \
   net-tools \
   python \
   python-dev \
   python-setuptools \
   python-distribute \
   python3-pip \
   tar \
   wget

# Get pip3 to download and install Python requirements:
RUN pip3 install flask
RUN pip3 install cherrypy
```

    d. The next commands copy the application folder to make it visible to the server, open access to port 80 for Web access and set the default directory in which the application will execute.

```
# Copy the application folder inside the container
ADD /app /app

# Expose ports
EXPOSE 80

# Set the default directory where CMD will execute
WORKDIR /app
```

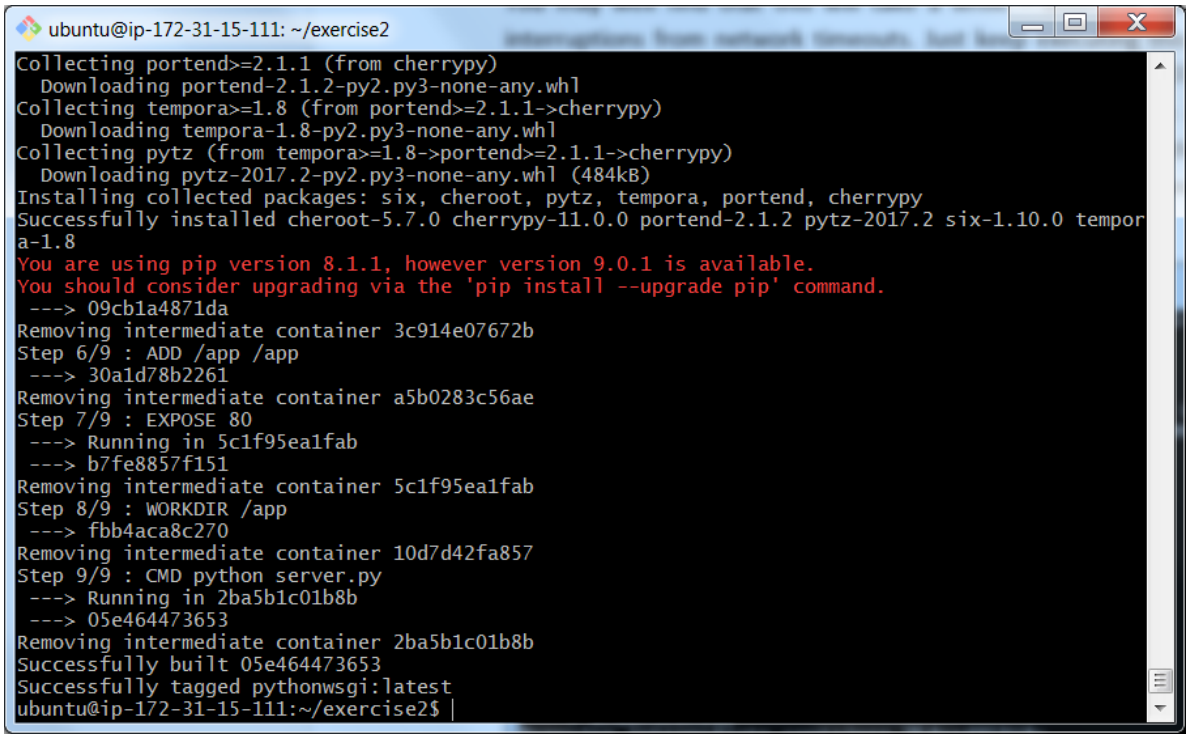
e. Finally the CMD command is used as the entry point for the application and it starts the server

```
# Set the default command to execute when creating a new container
# i.e. using CherryPy to serve the application
CMD python3 server.py
```

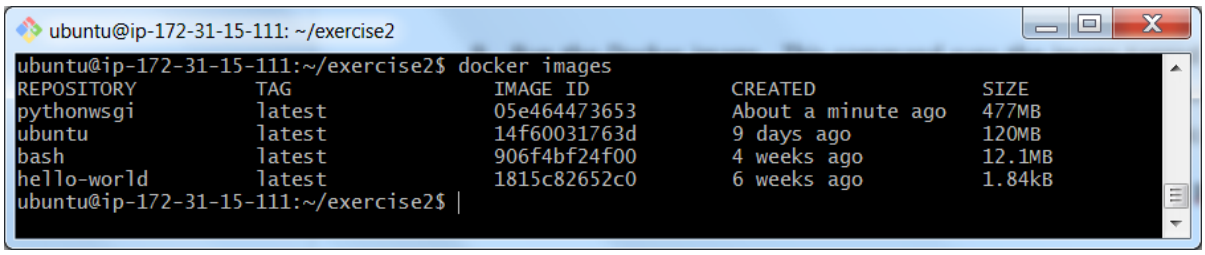
8. Build the Docker image. It will be tagged and referenced as `pythonwsgi`. This command looks for `Dockerfile` in the current directory i.e. '**.**' and builds the image according to the script which was just generated.

`docker build -t pythonwsgi .`

This may take a while – both the update process and the possible interruptions from network timeouts. Just keep executing this command until eventually you get a message that you have completed successfully. The `RUN apt-get` command is the most troublesome. Docker will create an intermediate container for each of the commands successfully executed and preserve these until the build is completed. The process will skip over completed steps and re-commence at the start of the next task, even if we have attempted it before.



The screenshot above shows the output I received after the final execution of the command. On this occasion I did not need to try multiple times. Your output may vary slightly. You can confirm that the image has been created using the `docker images` command:



9. Run the Docker image. This command runs the image tagged `pythonwsgi` in a Docker container which we name `exercise2`, and sets the correct port for external communications. The command line will wait until you press Ctrl + C to stop the application.

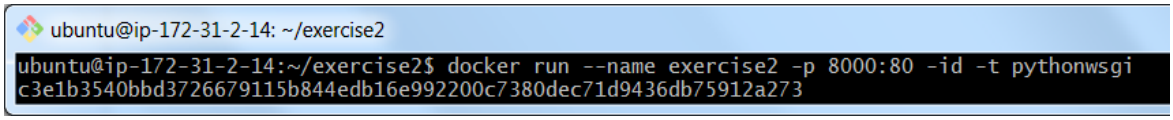
   `docker run --name exercise2 –p 8000:80 –i –t pythonwsgi`

   Here we are using the –p flag, which maps the machine and container ports:

   `-p local-machine-port:docker-container-port`

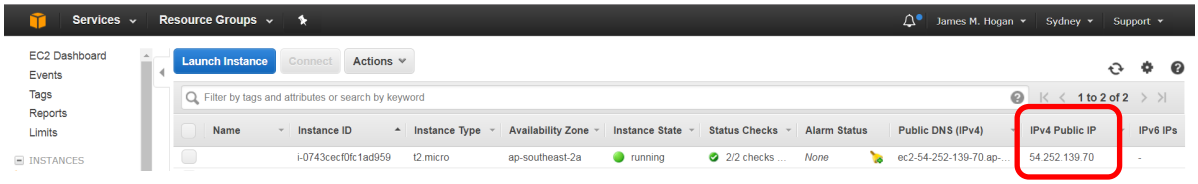
   Here we are mapping port 80 of the Docker container (the default for http) to port 8000 on the VM server. If we want to let this application run and continue working at the command line we can add the **–d** flag to daemonise the application:

   `docker run --name exercise2 –p 8000:80 –i –d –t pythonwsgi`

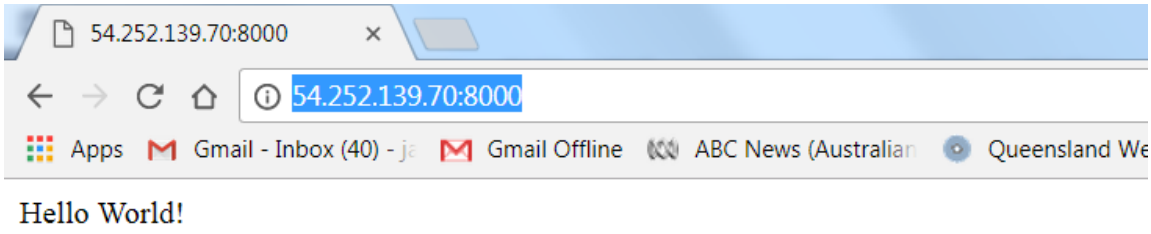


   The daemonised version will return a hash as shown.

10. To confirm that the application is in place and serving pages correctly, you should open a web browser and navigate to the VM IP address. Note that the machine name may tell you the IP address, but if this is from an earlier AMI (as in this case), the approach isn't reliable. So instead, check properly in the EC2 dashboard as shown. You could also use the full domain name from the connection information.



   In my case, I open a browser at: http://54.252.139.70:8000/ and see Hello World! as expected. Note that this is ***NOT*** port 80 as we have on the container. Port mapping works.



Exercise 3 – Ubuntu with Python Server (Word Count)

In this exercise you will build on the Docker image created in Exercise 2, install some extra Python libraries, and alter the Python app from Exercise 2 to download data from the Natural Language Toolkit (NLTK) data store, create a frequency distribution of the words in the data and display the most frequent words in HTML so that each word has a size and colour based on its relative frequency – see the example below. Again, you will be able to view the output in a browser by navigating to the IP address of the Docker machine.

NLTK is a Python library and relies on the installation of the `scipy` and `numpy` packages. An NLTK Cheatsheet with Python code for some common tasks exists at http://bet.andr.io/code/nltk/cheatsheet/

1. Navigate to your work area and create a new directory named `exercise3`. Copy the contents of the `exercise2` directory into the new directory.
2. Edit `Dockerfile`. The following libraries need to be installed using `apt-get python3-numpy` and `python3-scipy`. The NLTK library (`nltk`) has to be installed using `pip3` with a `-U` flag.
3. The major changes will occur in the `app.py` file.
    a. The following imports and downloads are required.

```
import nltk
from nltk import FreqDist

nltk.download('gutenberg')
from nltk.corpus import gutenberg
```

Project Gutenberg is a collection of books and part of the NLTK data collection.

b. We will now create a method named `count_words()` which will replace `hello()`. The method will load a book from Project Gutenberg, create a frequency distribution, extract the most common 500 (or some other number) tokens (tokens include punctuation and duplicate words with different case) and

generate HTML to display the most common words in different font sizes and colours depending on the word's relative frequency to the maximum frequency.

c. We first load the tokens from Sense and Sensibility, and then create a list of lower case words if they are not punctuation.

```
tokens = gutenberg.words('austen-sense.txt')
tokens = [word.lower() for word in tokens if word.isalpha()]
```

d. Create a frequency distribution using the extracted tokens. From the frequency distribution extract the most common 500 words.

```
fdist = FreqDist(tokens)
common = fdist.most_common(500)
```

e. The most common word data structure is a sequence with each entry being a tuple of the word and its frequency. To be able to create the word output we need to extract the words from the dictionary into a list and sort them.

```
words = []
for word, frequency in common:
    words.append(word)
words.sort()
```

f. We need to get the frequency of the most common word for formatting the font size and colour of the HTML output.

```
highCount = common[0][1]
```

g. Now that we have an alphabetically sorted list of the most common words we can start to build the HTML output. Declare a string variable named `html` and assign to it an opening `html` tag followed by a `head` with `title`, followed by an opening `body` tag. An `h1` heading can be added as well.

h. For each word in words we can now calculate a size in pixels to display and calculate a hexadecimal colour. The `hex` function converts an integer to hexadeximal but the result has 0x before the hexadecimal digits. They need to be removed and the hex needs to be padded with leading 0s to a length of 6.

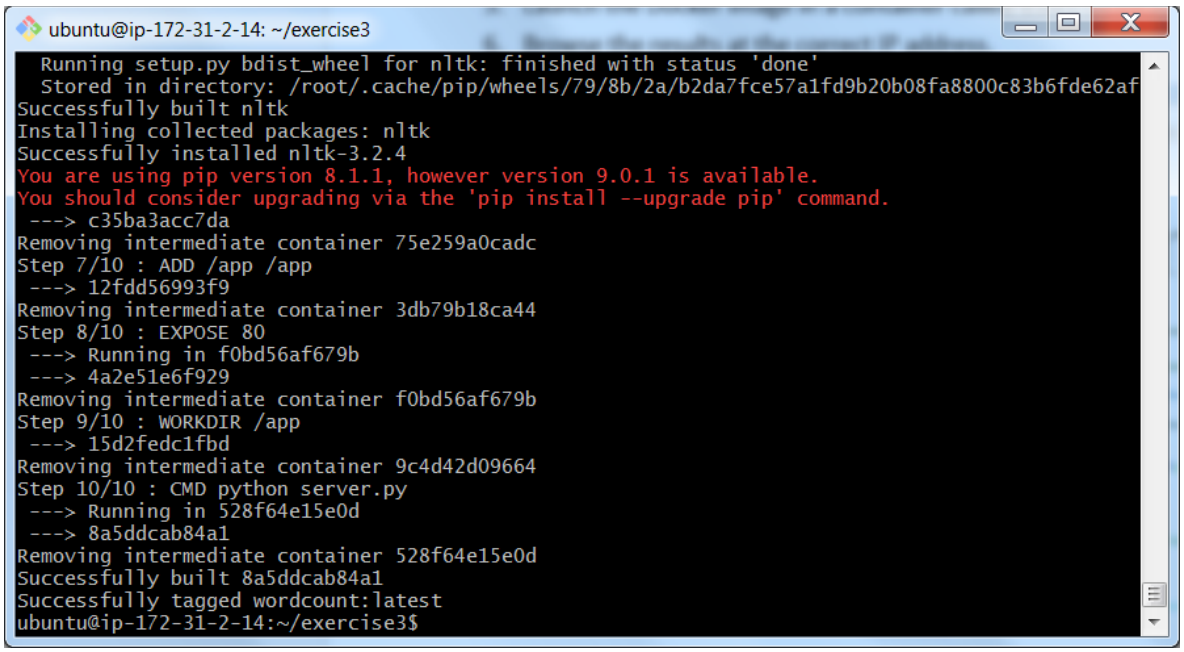
```
size = str(int(15 + fdist[word] / float(highCount) * 150))
colour = str(hex(int(0.8 * fdist[word] / \
                     float(highCount) * 256**3)))
colour = colour[-(len(colour) - 2):]
while len(colour) < 6:
    colour = "0" + colour
```

i. Each `word` can be added to a HTML `span`. Each span can be given CSS style for `font-size` in `px` and `color` which must be preceded by a `#`. An example span element is given below. Follow each span element with a space character so the HTML breaks across the page.

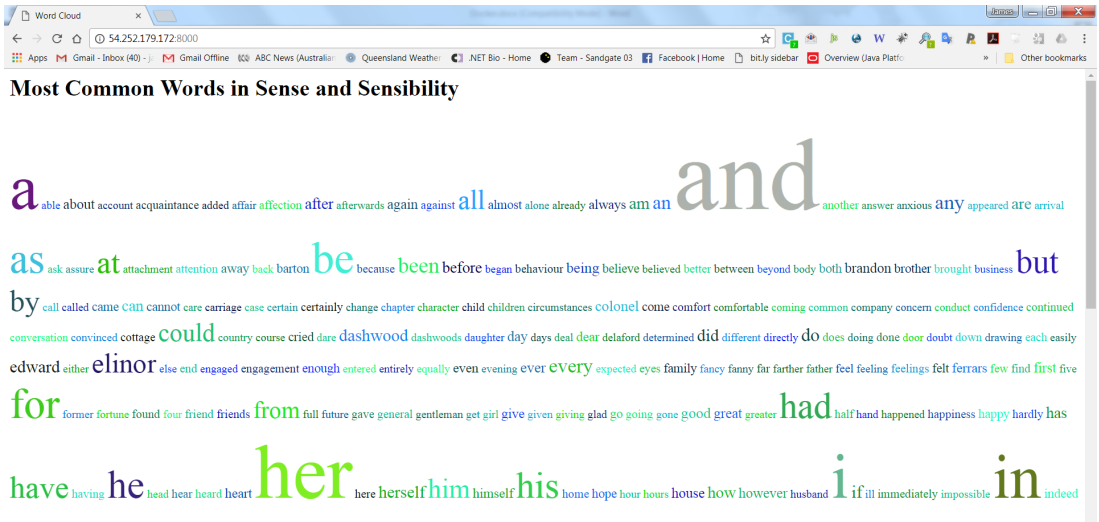
```
<span style="font-size: 16px; color: #017e22">comfortable</span>
```

j. After the `for` loop is completed we can add the closing `body` and `html` tags to the `html` variable and `return` it.

4. Build the Docker image `wordcount` using the same commands as before. After rather a long time, you will see something like the following:



5. Launch the Docker image (again modifying the commands you have seen above) in a container called `exercise3`. Browse the results at the correct IP address, and once again at port 8000, unless you have changed the security settings for the instance (this may not apply for local Linux machines and will depend on your security settings).



Exercise 4 – Word Count Without Stop Words

In this exercise you will build on the Docker image created in Exercise 3 by removing the *Stop Words* from the word list and displaying everything else. Just keep the same directory structure – we are just modifying `app.py` to make it more realistic. Stop Words in natural language processing are the most common words in a language. For English that includes: verbs like *is*, *are*, *be* etc.; articles like *a*, *an* and the; pronouns like *he*, *she*, *they* etc.; prepositions like *on*, *in*, *at* etc.; and others. The NLTK has a list of stop words that can be imported.

Simple guidance on using the stop word lists in NLTK may be found on the web at https://pythonspot.com/en/nltk-stop-words/. We steal the same ideas and apply them to the Jane Austen word list.

1. We first import the stop words – put these below the other imports

```
nltk.download('stopwords')
from nltk.corpus import stopwords
```

2. We then create a set of English stopwords – put this under the function definition

```
# Define the stopword set
stopWords = set(stopwords.words('english'))
```

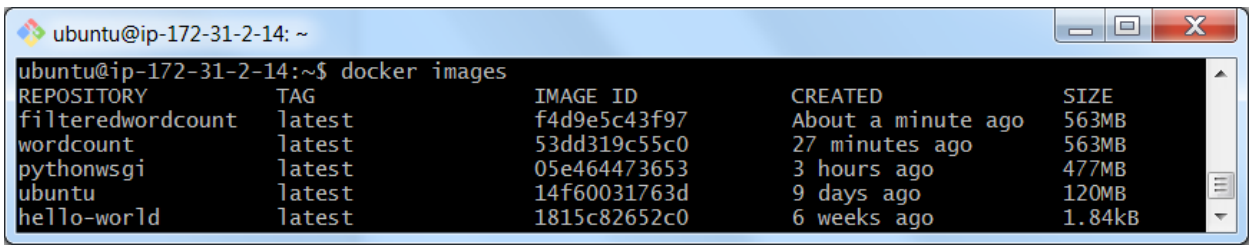
3. We then add an additional filtering line in the processing of Sense and Sensibility

```
# Grab Sense and Sensibility; tokenize; filter stop words;
# get frequency distribution
tokens = gutenberg.words('austen-sense.txt')
tokens = [word.lower() for word in tokens if word.isalpha()]
tokens = [word for word in tokens if word not in stopWords]
fdist = FreqDist(tokens)
```

After rebuilding the Docker image (perhaps you can call it `filteredwordcount`) we can run it in a container called `exercise4`. With the usual conventions we will see something like the image on the next page. It is a somewhat better reflection of the style of the writer.

5. Finalising – pushing to Docker Hub and pulling to another machine

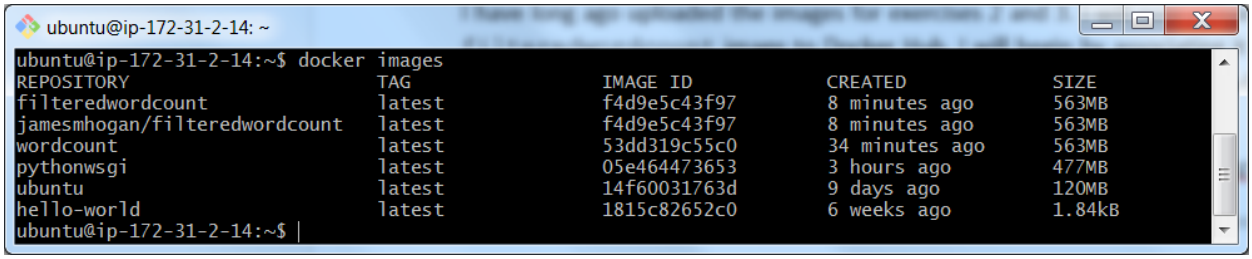
Earlier, following the tutorial from Docker, you created your own Docker Hub account. Using the Docker images command (and cleaning a few unwanted images), we have something like:



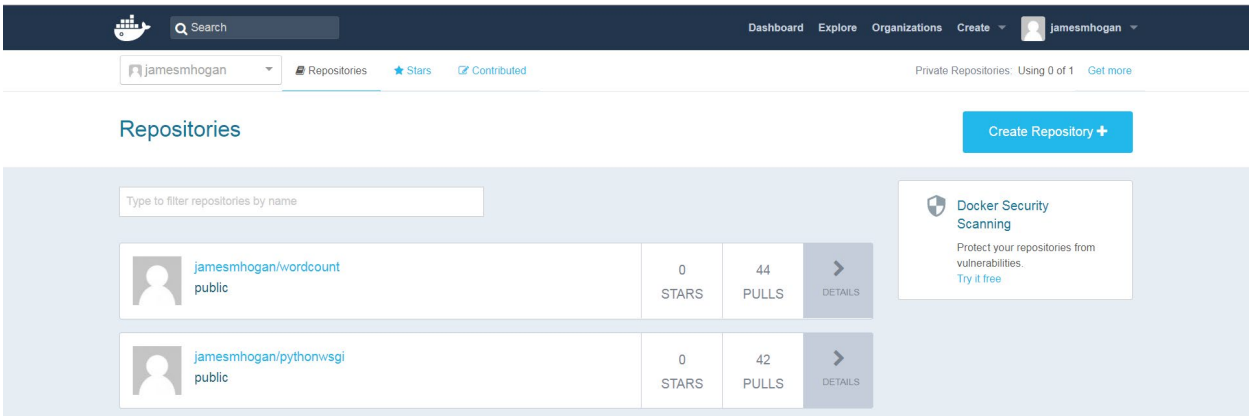
I have long ago uploaded the images for exercises 2 and 3. I will now upload the most recent `filteredwordcount` image to Docker Hub. I will begin by associating it with my account. This is handled by the tagging command, which may cover a rename as well. In my case, the command has the form (note the abbreviated ID):

```
docker tag f4d9e jamesmhogan/filteredwordcount:latest
```

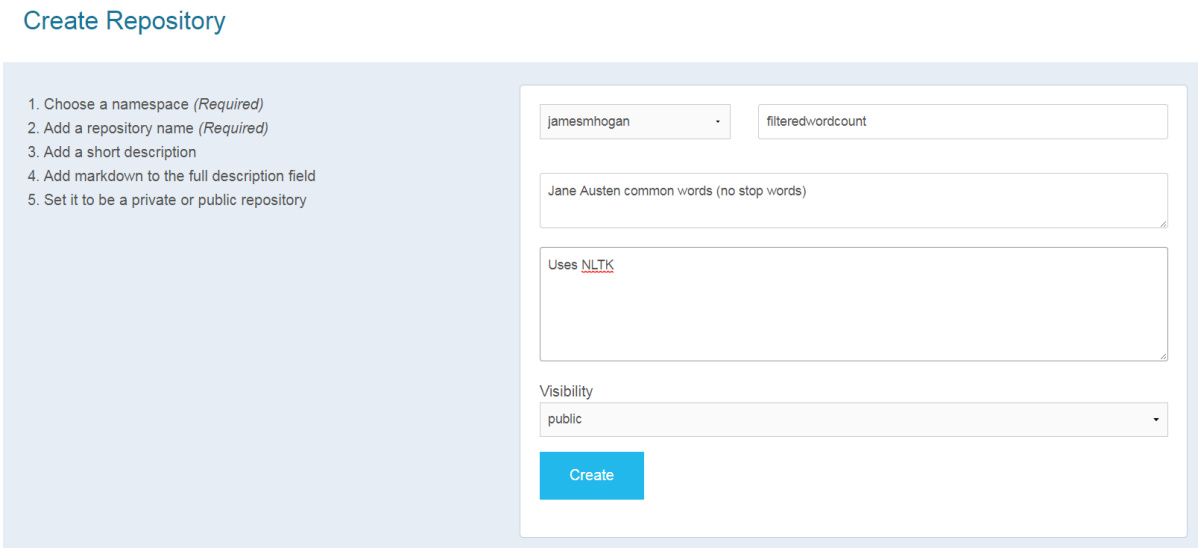
As before, we see the list of images available, with dual entries reflecting the multiple tags.



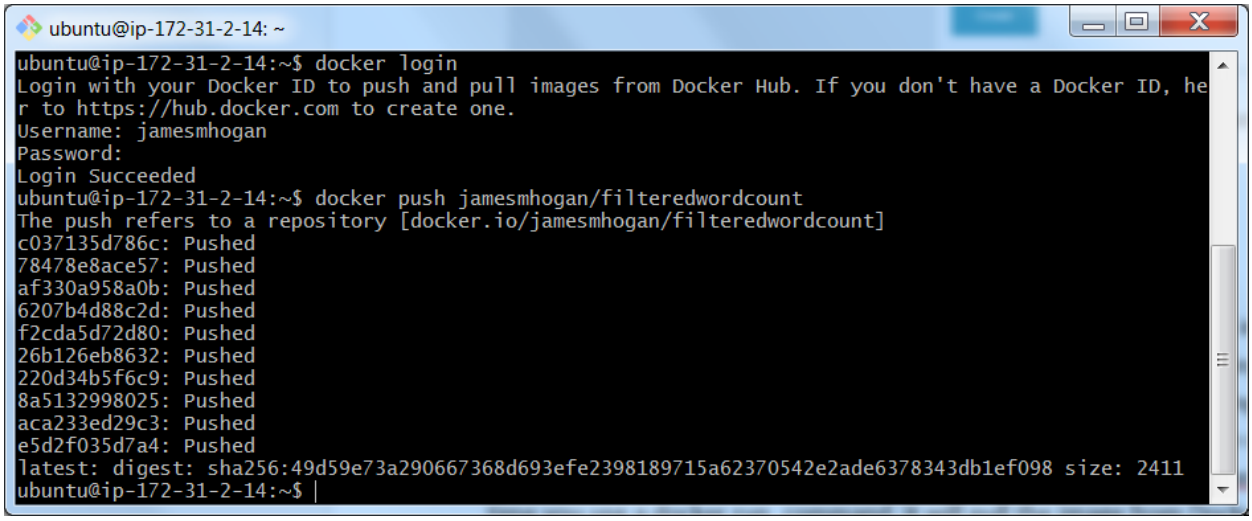
We follow the tutorial and upload our images. Currently, my dashboard on Docker Hub looks like this, showing the two public repos for exercises 2 and 3 above.



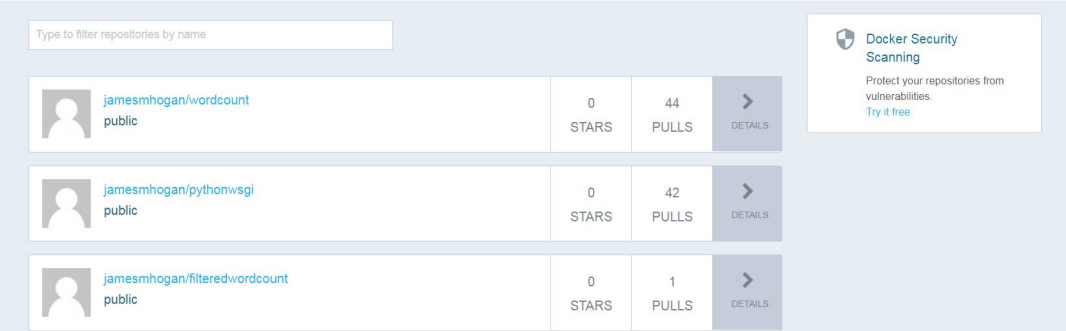
After clicking on the blue *Create Repository* button at top right, we have the following. I have added the appropriate repo name and a basic description. We now click on *Create*.



The image below shows a `docker login` and then a successful push to Docker Hub.

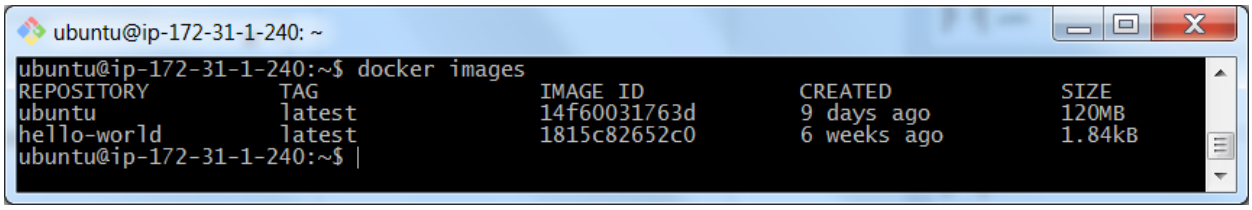


And here we see the revised version of the dashboard view:



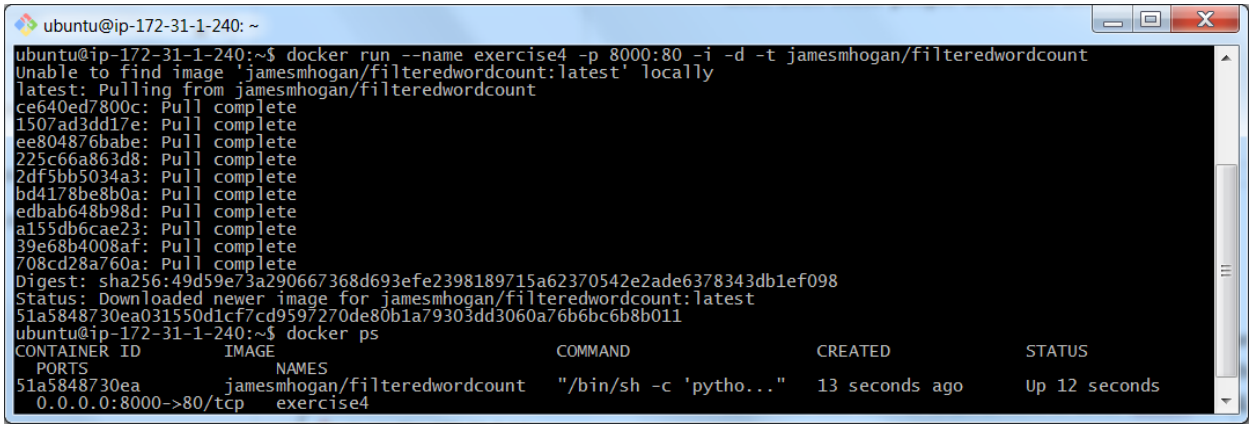
Now, as a final test of everything, fire up a second instance of the basic AMI which we started with. Even those people who have been working on a local Linux machine should do this. The

configuration should have a basic Ubuntu LTS 18.04 and a current installation of Docker. Your goal is to pull down the latest of your images and run the application that we have just created. After all, this sort of deployment is what Docker is all about. As before, expose a port other than the standard http port 80. In my case, I will again use 8000. We will again map the container port 80 to the server port 8000. At present, I am starting with just two basic images:



We go ahead and run the image:

```
docker run --name exercise4 -p 8000:80 -i -d -t jamesmhogan/filteredwordcount
```



And once again we see the filtered version of the wordcount in the browser:

That concludes the exercise. Make sure you have a local copy of the python files you have created, and/or create an updated AMI and preserve the machine state. Then go ahead and ***TERMINATE ALL EC2 INSTANCES*** as you have previously been instructed to do. As before, ***do NOT just STOP them.*** You must ensure that instances are terminated if you wish to avoid charges.

Appendix

Docker and Other Operating Systems

In some earlier versions of this unit we supported Docker under both Linux and Windows. However, we had substantial issues when using Docker under Windows 7 and so we focused on Linux. Docker support for Windows 10 is much better, and it is known also to work well in the Windows Subsystem for Linux – though there are configuration issues which mean this isn't the case for QUT lab machines. There are legacy solutions for Windows 7 if you still have it, but just don't do it. Use the cloud machines or the Virtual Box approach outlined below.

If you want to run Docker on your home machine running Windows or Mac OS then you should first download the installer at:

https://www.docker.com/products/docker-desktop

The site will detect your OS and select the right version.

You then need to learn a bit more about Docker on your system. The Docker documentation has a number of sections devoted to other operating systems. Those working on the Mac should look at:

https://www.docker.com/docker-mac

while Windows users should start at:

https://docs.docker.com/docker-for-windows/

Linux Command Line and Editors

There are many Linux tutorials on the web, and these are of varying quality. This one from Ryan's Tutorials seems a reasonable start. Others may suit you better.

http://ryanstutorials.net/linuxtutorial/

Commonly people use the editors vi and nano. Some also use emacs. Here are some basic tutorials to get you started.

http://heather.cs.ucdavis.edu/~matloff/UnixAndC/Editors/ViIntro.html

http://ryanstutorials.net/linuxtutorial/vi.php

https://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/

http://www.tuxradar.com/content/emacs-tutorial-beginners

https://www.gnu.org/software/emacs/tour/

Linux Under Virtual Box

Oracle's Virtual Box is a very widely used VM manager for Windows. It is very convenient and really doesn't take too much time to set up and use productively. The downloads are found at:

https://www.virtualbox.org/

When you look at the page, as shown above, you will also see a link to pre-built images. These are for Oracle's versions of Linux, so for consistency with our cloud images, do not click on this link. Instead, follow the download and installation instructions that you find in this blog below:

http://www.beopensource.com/2018/02/Install-Ubuntu-18.04-Virtual-Box.html