Tutorial

Docker

In this tutorial, you will learn the following:

- How to find and pull Docker images
- How to run a Docker container
- Understand the lightweight nature of docker containers
- How to create your own docker images that build on existing images

To complete this tutorial you will need to submit 4 screenshots to Canvas that you will take throughout the tutorial.

Purpose of Docker

The purpose of Docker is to put our application and our application dependencies into an isolated Docker container that we can then ship to our clients. The container can then be executed by anyone who has Docker installed on their machine. For example, if your application only runs on Linux, you could build a docker container on top of a container containing the Ubuntu OS so that your application could run on any operating system.

As another example, let’s say that we have an application that only works on Java 7. Without Docker we would need to have our users install Java 7 and then make sure that the user runs the application using Java 7 and not a different version of Java. However, with Docker we can put the Java 7 jdk in our Docker container along with our application so our users can simply run the Docker container without any setup.

Managing and Downloading Images

In this section you will learn how to find and pull Docker images. Docker images are used to create a docker container.

1. Open Terminal (or the quick start docker terminal for Windows users)
2. In order to create a Docker container, we will first need a Docker image. Docker provides many different images that we can find on hub.docker.com. We will be using two of these images during this tutorial.
    a. Ubuntu Image

   b. Openjdk Image

3. You should have installed the following two images as part of the topic preparation

   a. ubuntu
   b. openjdk

4. Make sure your images are installed by running the following command.

   a. `docker images`

5. Your terminal should list the images for ubuntu, openjdk, and any you pulled or ran previously.

6. **Take a screenshot of your images to submit to Canvas later.**

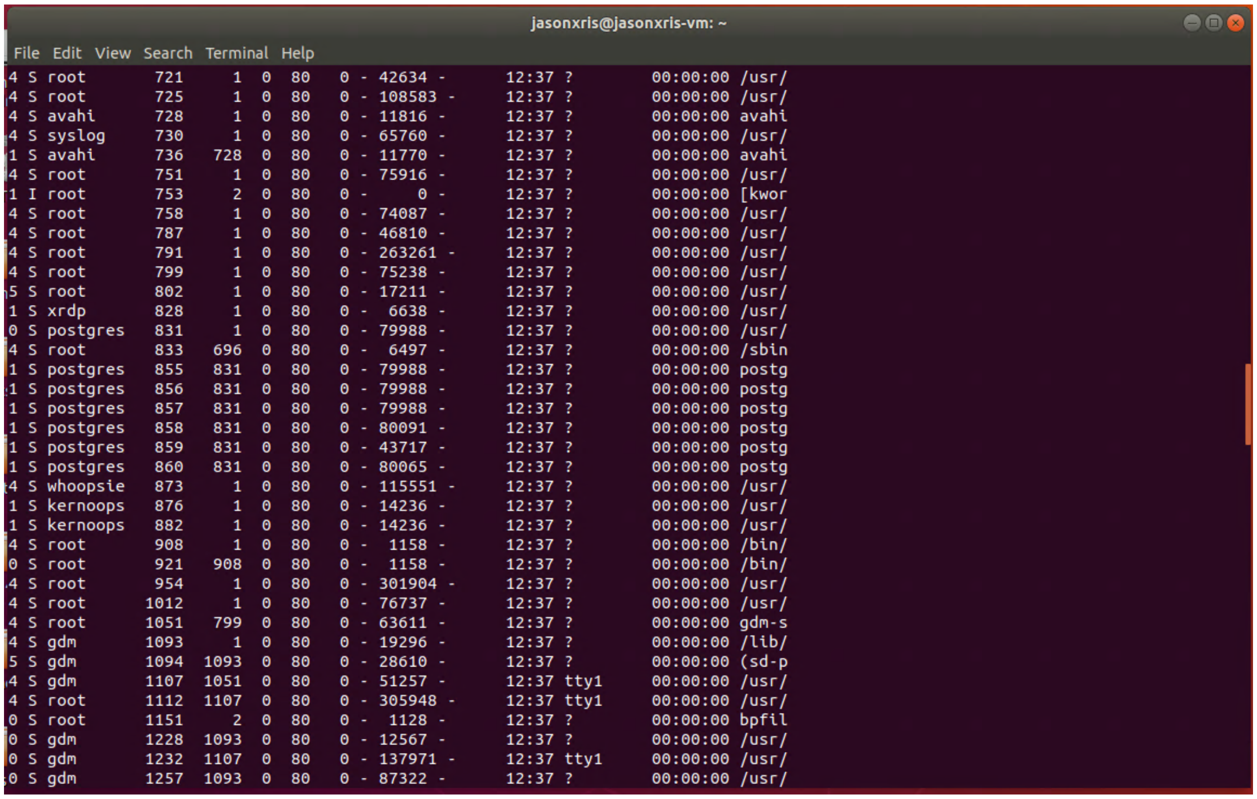
Creating a Docker Container

In this section you will learn both how to create a container and how the Ubuntu VM and the Ubuntu Docker Container differ. You will also learn basic Docker commands for containers.

1. A Ubuntu VM runs many processes. These are used to manage all of the functions of a fully fleshed out Operating system.
   Typing the following command in the terminal of a Ubuntu VM. reveals all of the processes that are currently running

   a. ps -elf

2. Here is a screenshot of what running that command looks like


jasonxris@jasonxris-vm: ~
File Edit View Search Terminal Help

```
4 S root        721     1  0  80   0 - 42634 -      12:37 ?        00:00:00 /usr/
4 S root        725     1  0  80   0 - 108583 -     12:37 ?        00:00:00 /usr/
4 S avahi       728     1  0  80   0 - 11816 -      12:37 ?        00:00:00 avahi
4 S syslog      730     1  0  80   0 - 65760 -      12:37 ?        00:00:00 /usr/
1 S avahi       736   728  0  80   0 - 11770 -      12:37 ?        00:00:00 avahi
4 S root        751     1  0  80   0 - 75916 -      12:37 ?        00:00:00 /usr/
1 I root        753     2  0  80   0 -     0 -      12:37 ?        00:00:00 [kwor
4 S root        758     1  0  80   0 - 74087 -      12:37 ?        00:00:00 /usr/
4 S root        787     1  0  80   0 - 46810 -      12:37 ?        00:00:00 /usr/
4 S root        791     1  0  80   0 - 263261 -     12:37 ?        00:00:00 /usr/
4 S root        799     1  0  80   0 - 75238 -      12:37 ?        00:00:00 /usr/
5 S root        802     1  0  80   0 - 17211 -      12:37 ?        00:00:00 /usr/
1 S xrdp        828     1  0  80   0 -  6638 -      12:37 ?        00:00:00 /usr/
0 S postgres    831     1  0  80   0 - 79988 -      12:37 ?        00:00:00 /usr/
4 S root        833   696  0  80   0 -  6497 -      12:37 ?        00:00:00 /sbin
1 S postgres    855   831  0  80   0 - 79988 -      12:37 ?        00:00:00 postg
1 S postgres    856   831  0  80   0 - 79988 -      12:37 ?        00:00:00 postg
1 S postgres    857   831  0  80   0 - 79988 -      12:37 ?        00:00:00 postg
1 S postgres    858   831  0  80   0 - 80091 -      12:37 ?        00:00:00 postg
1 S postgres    859   831  0  80   0 - 43717 -      12:37 ?        00:00:00 postg
1 S postgres    860   831  0  80   0 - 80065 -      12:37 ?        00:00:00 postg
4 S whoopsie    873     1  0  80   0 - 115551 -     12:37 ?        00:00:00 /usr/
1 S kernoops    876     1  0  80   0 - 14236 -      12:37 ?        00:00:00 /usr/
1 S kernoops    882     1  0  80   0 - 14236 -      12:37 ?        00:00:00 /usr/
4 S root        908     1  0  80   0 -  1158 -      12:37 ?        00:00:00 /bin/
0 S root        921   908  0  80   0 -  1158 -      12:37 ?        00:00:00 /bin/
4 S root        954     1  0  80   0 - 301904 -     12:37 ?        00:00:00 /usr/
4 S root       1012     1  0  80   0 - 76737 -      12:37 ?        00:00:00 /usr/
4 S root       1051   799  0  80   0 - 63611 -      12:37 ?        00:00:00 gdm-s
4 S gdm        1093     1  0  80   0 - 19296 -      12:37 ?        00:00:00 /lib/
5 S gdm        1094  1093  0  80   0 - 28610 -      12:37 ?        00:00:00 (sd-p
4 S gdm        1107  1051  0  80   0 - 51257 -      12:37 tty1     00:00:00 /usr/
4 S root       1112  1107  0  80   0 - 305948 -     12:37 tty1     00:00:00 /usr/
0 S root       1151     2  0  80   0 -  1128 -      12:37 ?        00:00:00 bpfil
0 S gdm        1228  1093  0  80   0 - 12567 -      12:37 ?        00:00:00 /usr/
0 S gdm        1232  1107  0  80   0 - 137971 -     12:37 tty1     00:00:00 /usr/
0 S gdm        1257  1093  0  80   0 - 87322 -      12:37 ?        00:00:00 /usr/
```

3. Notice that there are lots of processes running at the same time.

4. Now lets create an Ubuntu Docker container and compare the number of processes running in it.

5. Run the following command in the terminal to create an Ubuntu Docker container and run the bash program on it.

    a. docker run -it --name ubuntu-mini ubuntu /bin/bash

    b. This will create a docker container with the following properties

        i. Interactive (-it)
        ii. Named “ubuntu-mini” (--name ubuntu-mini)
        iii. From the “ubuntu” image (ubuntu)
        iv. Running bash (/bin/bash)

6. We are now in the bash shell for our Docker container

7. Run the following command in the “ubuntu-mini” bash (from the prompt you are now in)

    a. ps -elf

8. Notice the small number of processes compared to what you saw in a full running Ubuntu VM. This is because the ubuntu image creates the bare bones of the Ubuntu OS. This is important to know because the Docker Container cannot do as much as the Ubuntu OS. For example if you were to run the command `ping google.com` the Ubuntu Docker Container would not know the command “ping” while the Ubuntu VM would run the command. If you needed more commands, you would need to create a different image that contains them.

9. **Take a screenshot of the ubuntu-mini processes to submit to Canvas later.**

10. Exit from the ubuntu-mini bash shell.

    a. If running from Windows, do this by pressing **ctrl + q + p**.

    b. If running from macOS, do this by typing **control+p** followed by **control+q**

11. We can view our running containers by typing

    a. docker ps

12. You should see your “ubuntu-mini” container currently running.

13. Now type the following command to stop your container

    a. docker stop ubuntu-mini

14. Run the following command and make sure that there are no running containers.

    a. docker ps

15. To view containers that are currently stopped type the following:

    a. docker ps -a

    The -a indicates that we want to see all containers (not just the ones that are running)

16. Remove the ubuntu container we create above by typing the following:

    a. docker rm ubuntu-mini

Creating a Docker Image

In the pre-class reading, you learned how to download images someone else created and use them to generate and run containers. Now we will create our own Docker image from which we can create and run a container. We will create a container to run the exact same bash script we ran in our virtual machine tutorial last week. However, we will do it by creating a Docker image consisting of a light-weight version of Ubuntu and our bash script, and we will run it from Docker without needing the virtual machine we created last week.

1. Create a folder called ‘docker-hello204student’. The path to this folder must not contain any spaces.
2. Download into this directory the start bash script you used in the Virtual Machine tutorial.
   a. Make sure to **download the Start file** instead of copy and pasting it into a new file.
      i. This is because Windows and Linux have different end-of-line symbols. So your copy and paste script will not work on the docker linux container
3. Create a file in this directory called Dockerfile (with no file extension).

   Note: A Dockerfile is an instruction file that tells Docker how to build a Docker image. We will learn a few basic instructions needed to build some simple images. The complete syntax is much more than we can learn in this tutorial.

4. Type the following commands into your Dockerfile file:

```
from ubuntu
ADD ./start /
RUN chmod +x start
ENTRYPOINT ./start
```

   Each of the commands is described below:

   - The ‘from ubuntu’ command says that we want to build an image on top of the existing ubuntu image. If the ubuntu image does not already exist on our machine, Docker will download it when it builds an image from our Dockerfile.
   - ‘ADD ./start /’ tells Docker to copy the ‘start’ file from the directory containing Dockerfile into the root ‘/’ directory of the Ubuntu based image we are creating.
   - The ‘start’ script we are coping into the image needs to have it’s executable (x) attribute set before it can be executed on the image we are creating. The ‘RUN chmod +x start’ command makes the file we added to the image executable.
   - ‘ENTRYPOINT ./start’ makes our new image executable and makes it so it will execute the start script when we run the image. There are multiple commands that can be used to tell an image to execute something. ENTRYPOINT is a common way to do it.

5. With the bash script (start file) in the directory and the Dockerfile created, we are ready to generate the image. Do this by executing the following command from the directory that contains your Dockerfile:

```
docker build -t hello-204student .
```

Don’t forget the dot at the end of the command. It is required and means to build the image from the current directory.

6. Run the image with the following command:

   ```
   docker run hello-204student
   ```

7. This does almost what we want, but notice that the script doesn’t wait for our input before printing the result string. This is because we didn’t run it in a way that allows the image to interact with the terminal. Fix that by running the image with the -i flag as follows (type your first and last name when prompted):

   ```
   docker run -i hello-204student
   ```

8. **Take a screenshot of the terminal output. You will submit this screenshot at the end of the lab.**

Pushing Your Docker Image to Docker Hub

Now that you’ve created a Docker image, you can share it (publicly or with a few users you select) by posting it to docker hub.

1. Start by navigating to Docker Hub.
2. If you already have a docker hub account, login. If not, create a new account.
3. Before we can push our image to docker hub, we need to tag it. This gives it the name by which it will be known on docker hub. The name starts with your docker hub username, followed by a slash ‘/’ and the image name. We also need to find the Image ID to use in the tag command. Find the image ID by executing the following command:

   ```
   docker images
   ```

4. Now tag the image with the following command (substituting your Image’s image ID for the number after the word ‘tag’ and your Docker Hub user name for ‘username’):

   ```
   docker tag cd92b3a26165 username/hello-204student
   ```

5. With our image tagged, we are almost ready to push it to docker hub, but first, we have to login from the command-line / terminal. Do that with the following command:

```
docker login -u=<your docker hub username> -p <your docker hub
password>
```

Note: This is the easiest way to login, but Docker will warn you that this is insecure. For a more secure way to login, you can use the `--password-stdin` flag but you have to then provide your username with OS specific syntax for sending the password to the command from a password file.

6. After you have successfully logged in, you push your image to Docker Hub with the following command (substituting your Docker Hub username for ‘username’):

   ```
   docker push username/hello-204student
   ```

   Wait a few seconds and you should see a new public repo appear in your Docker Hub account. This can be downloaded and executed by anyone who knows the repo name.

Using Docker to Run a Java Program Without Installing a JDK on the Host Machine

The purpose of a Docker container is to contain all of the necessary pieces to run our application. For example, we can use a Docker image to compile and run a Java program even if we don’t have Java installed on our machine. In this section we will build a Docker image on top of the openjdk Docker image to compile and run a simple Java Hello World program. You likely already have Java installed on your host machine, but this example has no dependencies on any previously installed Java JDK or JRE and would work even if you didn’t have Java installed.

1. Create a folder called “docker-java-hello-world” on your computer. The path to this folder must not contain any spaces!
2. Copy Main.java into the docker-java-hello-world folder.
3. Create a Dockerfile file in the docker-java-hello-world with the following contents:

   ```
   from openjdk
   ADD ./Main.java /
   RUN javac /Main.java
   ENTRYPOINT ["java", "Main"]
   ```

   Most of the commands should be familiar from the previous example. In this example, we are building an image on top of the ‘openjdk’ image which already contains an operating system and a working Java jdk. The ‘RUN javac /Main.java’ command compiles the java source file after it is copied into our new image and the 'ENTRYPOINT' command invokes the jvm, passing Main as the argument.

4. Create the image by executing the following command from the directory that contains the Dockerfile file:

   ```
   docker build -t java-hello-world .
   ```

5. Now create and run a container from this new image by executing the run command:

   ```
   docker run java-hello-world
   ```

6. You just compiled and ran a Java program without requiring Java to be installed on the host machine!

7. **Take the last screenshot of your java application running in the terminal and turn in all four screenshots to complete the assignment.**