DevOps – Laboratory 4

Containerization and Docker

This lab will introduce you to containerisation, specifically using Docker. Docker allows you to create, deploy and run applications in packaged up blocks called containers. The container will consist of everything you need to run the application, eliminating the possibility of dependency or library conflicts.

# Getting Started

You will need to launch an AWS EC2 Instance on which you will complete this Lab. Use the guidance in Lab 1 along with the note below to do this now.

At the Configure Security Group stage of your instance configuration you will need to open another port that you will use late in this Lab. To do this, click on the ‘Add Rule’ button, set the Pot Range to 80 (as this is the port you will use later) and set Sources to Anywhere using the dropdown menu. Your configuration should look like this:

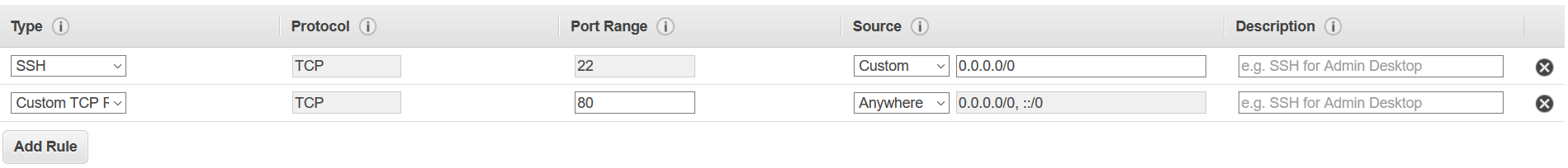


Figure 1: Opening specific ports

When you have successfully connected to your new VM via SSH you are ready to continue with this Lab.

**PLEASE REMEMBER TO STOP OR TERMINATE YOUR VM AT THE END OF THIS LAB SO THAT YOU DO NOT USE UNNECCESARY CREDIT!!**

# Important Information

This Lab documents will contain commands that you should run in the bash terminal (both locally and on your VM). This code will be provided in boxes as shown below:

|  |
| --- |
| This is a sample command that fits on one line  This is a much longer sample command that covers multiple lines but should still be copied as a single command |

As you can see, some commands will be short and clearly displayed on a single line. This can be copied directly into the Terminal. Other commands such as the second one are much longer and do not fit on one line. If you are going to copy and paste these commands make sure to copy all lines and paste as a single command. Each new command is separated by a blank line so that each command is clearly distinguished.

**If you are unfamiliar with using the Terminal or Bash it is highly recommended that you complete the following Tutorial:**

[**https://ubuntu.com/tutorials/command-line-for-beginners**](https://ubuntu.com/tutorials/command-line-for-beginners)

This tutorial is written for Ubuntu but as the Terminal we are using (GitBash for Windows or Terminal for MacOS/Linux) also uses Bash, the commands are the same and the tutorial is very relevant. The Virtual Machines you create on AWS will utilise the Ubuntu Operating system and therefore this tutorial will also be useful when interacting with your VMS.

Section 7 of this Tutorial covers SuperUser which is not applicable to Windows Systems but will be important on your VMs so reading is recommended.

The following Cheat Sheet is also helpful for remembering Bash commands and will be a good point of reference throughout this Module:

<https://cheatography.com/davechild/cheat-sheets/linux-command-line/pdf/>

**If you encounter any issued during this process, please contact the Module Tutor.**

# Installing Docker

From this point in the Lab it is assumed that you have created an AWS EC2 VM and have connected to it via SSH. The remainder of this Lab you should perform all command line tasks on the VM. If you have not already done this, please see the information earlier in this Lab.

Docker can be installed through the Package Manager on Ubuntu but you will first have to add the repository to the Package Manager before installation. The commands below (excluding the first line) can be used to install Docker.

**Keep reading and don’t run these yet but take some time to understand the commands below - you may need to use google etc.**

|  |
| --- |
| #! /bin/bash  sudo apt-get update  sudo apt-get -y remove unscd  sudo apt-get -y install apt-transport-https ca-certificates curl gnupg-agent software-properties-common  curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -  sudo apt-key fingerprint 0EBFCD88  sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"  sudo apt-get update  sudo apt-get -y install docker-ce docker-ce-cli containerd.io  sudo groupadd docker  sudo usermod -aG docker $USER  sudo addgroup --system docker  sudo adduser $USER docker  newgrp docker |

Rather than having to run these commands one at a time we can make use of Bash Scripting to run the commands in bulk. Line one above tells the OS that we want to utilize bash for the remaining lines in our script, after which we can list any standard bash commands which will be executed sequentially, exactly as though you had entered them into the terminal.

1. Use nano to create a file containing the bash script above (copy and paste the text from the box above in its entirety) called install\_docker.sh
2. Change the file permissions of the script using the chmod command as follows:

|  |
| --- |
| chmod 777 install\_docker.sh |

Here you are using the 777 flag to grant full Read, Write and Execute permissions to the file. For other combinations of permissions see the following link: <https://www.december.com/unix/ref/chmod.html>. You can also use online tools like this one (<https://chmod-calculator.com/>) to work the correct flag out for you.

1. Now you are ready to run the script and install docker. As the script has been made executable (chmod above) you can run this by simply entering the path to the file in the command line. Assuming this is in your current directory (i.e. you can see it when you run the ls command) you can use the following below.

**Note that in unix, a single period (.) indicates the current directory and double (..) indicates one directory previous. So assume you are in the directory /home/Ubuntu/DevOps/Lab1 and you run the command cd .. you would be taken to directory /home/Ubuntu/DevOps.**

|  |
| --- |
| ./install\_docker.sh |

1. Check that Docker has been installed correctly. Review the output and make sure there are no obvious errors and then use the following command and ensure that the output is similar to the example below.

|  |
| --- |
| docker version |

Output:

|  |
| --- |
| Client: Docker Engine - Community  Version: 19.03.2  API version: 1.40  Go version: go1.12.8  Git commit: 6a30dfc  Built: Thu Aug 29 05:26:49 2019  OS/Arch: darwin/amd64  Experimental: false  Server: Docker Engine - Community  Engine:  Version: 19.03.2  API version: 1.40 (minimum version 1.12)  Go version: go1.12.8  Git commit: 6a30dfc  Built: Thu Aug 29 05:32:21 2019  OS/Arch: linux/amd64  Experimental: false  containerd:  Version: v1.2.6  GitCommit: 894b81a4b802e4eb2a91d1ce216b8817763c29fb  runc:  Version: 1.0.0-rc8  GitCommit: 425e105d5a03fabd737a126ad93d62a9eeede87f  docker-init:  Version: 0.18.0  GitCommit: fec3683 |

**If the Server Information above is not displayed or gives a Permission Denied error, you should run the last 5 commands in the script manually one at a time and try the docker version command again. This issue occurs from time to time and appears to be a group permission error affecting a small number of users.**

# First Docker Container

1. Now you can run your first Docker container:

|  |
| --- |
| docker run hello-world |

If you study the output of this command, you will see that the first thing Docker does is check to see if the ‘hello-world’ image is available locally. As this is the first time you have used Docker, the image will not be available so Docker will automatically download it for you before running it. The reminder of the output is the ‘hello-world’ container running. All that this particular container does is print some information to the terminal but you will explore some more complex containers during this Lab.

1. Try running the above command again and look for any difference in the output from the first time.

Docker containers come in many shapes and sizes and can be run in three main ways which we will see in this section of the Lab. These are:

* Single Task: The container executes a specific command or application and then is shut down.
* Interactively: The container runs until it is told to stop and can be controlled by the user.
* Background: The container runs a service in the background such as hosting a website or running a monitoring app.

# Single Task Containers

In this section you will use single task containers to familiarize yourself with some of the basic Docker commands.

1. Run the command below and look closely at the output.

|  |
| --- |
| docker container run alpine hostname |

The above command tells Docker to use the alpine image to run the command ‘hostname’. You can see from the output, the image could not be found locally as you have never used it before. More importantly, docker is looking for ‘alpine:latest’ where ‘alpine’ is the image name and ‘latest’ is the tag. The tag will always default to ‘latest’ unless you specify otherwise and can be used to pull a specific version of an image if required.

If you look at the Docker Hub page for alpine you can see that there are many versions you could use. Have a look: <https://hub.docker.com/_/alpine?tab=tags>

1. Run a command on a different version of alpine:

|  |
| --- |
| docker container run alpine:edge ps |

You will see that Docker has to download the image as you are using a different version as specified by the ‘edge’ tag.

1. Run the following commands and look closely at the output of each:

|  |
| --- |
| docker image ls -a  docker ps -a  docker ps -l |

These commands allow you to; see all of the images that you have stored locally, see all of the containers you have run and see the last container you have run. These commands are useful as they also provide key information about each of the images/containers such as their IDs and names which you will need to use as you progress through this Lab.

1. Images which have been downloaded take up hard drive space which is often limited when using Cloud VMs. As it is no longer required, remove the alpine:edge image using the command below replacing image\_id with the ID of the appropriate image.

|  |
| --- |
| docker rmi image\_id |

**This command will result in an error** because the image has been used to create a container. To delete the image, you will first need to delete any containers that have been created using it.

1. Use the below command to delete any containers that use the alpine:edge image, replacing container\_name with the name of the appropriate container. You can get the container name from the commands at the top of this page and docker usually provides a bit of an unusual name when it creates a container.

|  |
| --- |
| docker rm container\_name |

1. Now go back and delete the alpine:edge image.
2. Run the following commands using the alpine image, have a look at the results and take note of what each command is doing.

|  |
| --- |
| docker run alpine ls  docker run alpine uname -a  docker run alpine uptime  docker run alpine free |

1. Now try the following Docker commands and take a note of what they are doing. These will be useful for you later.

|  |
| --- |
| docker ps -l  docker ps -a  docker ps -n 2 |

# Interactive Containers

Now you will explore containers that can be used for more than just a single task. Have a look at the command below:

|  |
| --- |
| docker container run --interactive --tty --rm ubuntu bash |

Three new components have been introduced:

--interactive - makes the container interactive

--tty - gives us a terminal to work with

--rm - removes the container after it has run

**Note: Had you used the --rm command previously you would not have had to manually delete the container before deleting the image as it would have been automatically deleted.**

1. Run the above command to launch the bash command in an interactive container based on the Ubuntu image.

You will notice that this time Docker downloads the image in multiple files called layers. These layers are individual and one layer can be used in multiple images, e.g. the next Ubuntu image release may only have changes to one of these layers and therefore when we download it the first time, Docker will only pull the new layer and reuse the old layers.

You should also notice that you are no longer interacting with your VM. The terminal prompt will have changed to ‘root@<container\_id>:/#’ which is the terminal created with the --tty command. Any commands that you enter now are being executed by the Ubuntu container that you are running.

1. Try out some commands and see how they work:

|  |
| --- |
| ls / - lists the root directory  ls -la - lists the current directory and hidden files  ps aux - shows all running processes  cat /etc/issue - shows the current OS info |

As you will see from the above stage, the container behaves exactly as you would expect a standard Linux installation to behave but you did not have to do a full installation. This can be very useful when using other operating systems such as Windows or MacOS.

To exit the container simply type ‘exit’ and press enter. As you used the --rm command this container will be deleted and you will not see it in the container list but the image still exists so you can easily launch a new container that will be identical to the previous.

If you wanted to install specific packages in this Ubuntu container and then use them later, it is possible to commit a container to create an image from it. This is not the best way to do this though and we will look at the alternative later which is the use of Dockerfile which can be used to build a repeatable, configurable image.

For a better example of when this type of container could be useful, imagine you have a small piece of code you would like to test.

Have a look at the code below:

|  |
| --- |
| public class AsciiValue {  public static void main(String[] args) {  char ch = 'D';  int ascii = ch;  // You can also cast char to int  int castAscii = (int) ch;  System.out.println("The ASCII value of " + ch + " is: " + ascii);  }  } |

1. Create a new file using the touch command called AsciiValue.java, open the file using nano, paste the above code into the file and save the changes.

The commands to compile and run this code are as follows:

javac AsciiValue.java

java AsciiValue

If you try this now you will see that it fails as Java is not installed on your VM. You could install the required packages (as in the Lab last week) but this takes some time and you may not want to have to install new packages to just run a quick test.

Instead you can launch a pre-configured Docker container and use that to compile and run the code.

1. Run the command below:

|  |
| --- |
| docker container run --interactive --tty --rm --mount type=bind,src=$(pwd),dst=/mnt openjdk bash |

You will notice two differences to the last time we ran the container run command. First, this command uses the openjdk image rather than Ubuntu. This is because we want to access the Java JDK which is part of this image. The second difference is the addition of the --mount flag:

--mount src=$(pwd),dst=/mnt, type=bind

This flag will mount a directory on the EC2 Instance to a directory inside the Docker container so that you can access the code you just created. This flag has three components:

src=$(pwd)

This sets the location on the EC2 that you want to mount. Here we are using $(pwd) which returns the current path.

dst=/mnt

This is the location within the container that we will connect to the src on the EC2. /mnt is a relatively standard location for external filesystems in lunix and is present on most linux systems which is why we have used it here.

type=bind

This sets the type of connection. Bind means that the locations are mirrored so any changes on one side are reflected on the other.

1. If you navigate to /mnt and see what is there you will notice the effects of this --mount flag as you should be able to see the AsciiValue.java file you created on the EC2 instance earlier.
2. Now run the Java compile and run commands from earlier and notice that because this container has the required libraries pre-installed, the commands now work and allow you to test the code.

Exit this container in preparation for the next section of the lab.

# Background Containers

Rather than running a single command or interacting with a container, you may want to run a container in the background so that it is delivering a service, such as a website or application, but we do not have to be actively engaging with it. This is how the majority of Docker containers are run.

1. Create a new container using the MySQL image using the command below:

|  |
| --- |
| docker container run --detach --name mydb -e MYSQL\_ROOT\_PASSWORD=my-secret-pw mysql:latest |

Again, some new parameters have been introduced:

--detach - runs the container in the background

--name - names it mydb

-e - uses an environment variable to set the root password (NOTE: This should never be done in production for security reasons)

1. Have a look at the containers that are currently running:

|  |
| --- |
| docker container ls |

1. You can also see what is happening inside the container using:

|  |
| --- |
| docker container logs mydb  docker container top mydb |

Logs will provide you with lots of information that we are not going to spend time on at the minute.

Top is much more helpful and shows any running processes within the container, which in this case is just the mySQL daemon (mysqld).

While this container is not interactive in the sense that the previous container was, we can still interface with it and run commands within it using the ‘exec’ command.

Imagine you would like to run the following command inside the container:

‘mysql --user=root --password=$MYSQL\_ROOT\_PASSWORD --version’

1. You can do so using the following command:

|  |
| --- |
| docker container exec -it mydb mysql --user=root --password=$MYSQL\_ROOT\_PASSWORD --version |

Here you are using the -it command, which is an abbreviated version of the commands --interactive and --tty that you used previously.

1. Try the below command which is similar but actually lets you interact with the container rather than just returning a value

|  |
| --- |
| docker exec -it mydb sh |

You should notice that the command prompt has changed because you are now connected to the sh process running inside the container. SH is a subset of the Bash shell you used previously. You can run the command from the previous step here directly without using the ‘exec’ command (because you are interacting with the sh process INSIDE the container) which should return the same value as follows:

|  |
| --- |
| mysql --user=root --password=$MYSQL\_ROOT\_PASSWORD --version |

To exit the interactive sh process type the ‘exit’ and press enter. You should now see the command prompt for your VM again.

You will see more examples of this type of container in the next section of this Lab.

# Custom Applications in Docker

In this section you will learn how to build and package your own application in Docker using a Dockerfile. You will deploy your own NGINX website and deploy it as a Docker container.

For this section of the Lab you will need your own DockerHub account. DockerHub is an online repository for Docker Images in a similar way that GitHub works for code etc.

1. Sign up for a DocerHub account here and make sure that you know your username and password.

<https://hub.docker.com>

1. Log into Docker via your Terminal by running the command below and following the onscreen prompts:

|  |
| --- |
| docker login |

1. Now set an environment variable containing your Docker ID replacing docker\_id below with the one you just created. This is not a mandatory step but will make many of the commands in this section much easier but please note that the environment variable will be cleared if you disconnect from the VM or close the terminal so you will have to set it again under these circumstances.

|  |
| --- |
| export DOCKERID=docker\_id |

1. Check that this has worked by running the command below. The result should be your Docker ID.

|  |
| --- |
| echo $DOCKERID |

You are now ready to start building a Docker Image from scratch.

1. Clone the following Git repository and navigate into the directory:

git clone <https://github.com/dockersamples/linux_tweet_app>

1. Have a look at the Dockerfile (nano Dockerfile) which in this case has four main commands:

FROM - sets the base image that will be used

COPY - copies files from the host into the image

EXPOSE - opens the required ports

CMD - sets the command that runs when the container starts as well as its arguments

1. You can use this Dockerfile to build a Docker Image, which can later be deployed as a container, using the following command:

|  |
| --- |
| docker image build --tag $DOCKERID/linux\_tweet\_app:1.0 . |

The --tag command allows you to give the image a name. Using your Docker ID as part of the tag allows you to publish your image to Docker Hub later.

The . at the end of the command specifies the directory that Docker should use for the build. A single period like this indicates the current directory. This must be the location of the Dockerfile you want to use or Docker will not be able to find the instructions for creating the image. You could also provide a full path to a Dockerfile in another location but it is easier to navigate to the correct folder.

1. Now launch a container using your newly created image:

|  |
| --- |
| docker container run --detach --publish 80:80 --name linux\_tweet\_app $DOCKERID/linux\_tweet\_app:1.0 |

Here, --publish 80:80 connects port 80 on the container to port 80 on the VM. You have already set port 80 on the VM to be open during the setup so now you should be able to connect to your website from anywhere using the Public IP address of the VM and the specific port.

1. Open a web browser on your local machine and in the address bar enter the following, replacing ip\_address with the public IPv4 address of your VM:

ip\_address:80

1. Once you have seen your website running you can stop and remove the container using the following commands:

|  |
| --- |
| docker container stop linux\_tweet\_app  docker container rm linux\_tweet\_app |

# Advanced Docker

In this section you will learn how to use some of the more advanced features of Docker to help you further manage the website you launched previously.

Until now, containers have run in relative isolation from the host machine. Here you will learn how to mount a directory from the host to the Docker container. This means that you will be able to modify the files in this directory and see the changes immediately reflected inside the container.

1. Run your container again exactly as before (previous step h) but this time include the following mount flag. Make sure you do this with the other flags starting with -- and not at the start or end of the command:

|  |
| --- |
| --mount src=$(pwd),dst=/usr/share/nginx/html,type=bind |

Notice that we are mounting the current directory (indicated by $(pwd) which should be the linux\_tweet\_app directory) to the directory inside the container which contains the HTML files for the website as defined in the Dockerfile.

1. Check that the website is running as you would expect before continuing.

Now that you have access to the files that Docker is using to host your website, you can make changes to the site while it is running.

1. In the linux\_tweet\_app directory on your VM, replace index.html with index-new.html as follows:

|  |
| --- |
| cp index-new.html index.html |

1. Now refresh your webpage to see the changes that have been introduced.

Even though you have made the changes to the index.html on the EC2 VM filesystem, the changes have not been permanently made to the image as it was built using the original file.

1. To prove this, stop and remove the currently running container and start it again using the original command without the --mount option. If you refresh the website again you will see that it has gone back to the original version.
2. Stop and remove this container. While we have previously done this with two commands, the following will do the stop and remove simultaneously:

|  |
| --- |
| docker rm --force linux\_tweet\_app |

1. Now you can create a new version of your image using the updated index.html:

|  |
| --- |
| docker image build --tag $DOCKERID/linux\_tweet\_app:2.0 . |

1. Now have a look at the images you have available (docker image ls) and you should see both versions of your app with Tags 1.0 and 2.0 as expected.
2. Run your new image and check your website to ensure that it shows the updated version as you would expect.

Finally, you can publish your own image to DockerHub so that you can access it from any machine and share it with other developers and users.

1. List only the images that you have created using the following command:

|  |
| --- |
| docker image ls -f reference="$DOCKERID/\*" |

These images are currently only available on the VM which you will delete at the end of this lab and would therefore lose any data stored on it.

1. You can push your image to DockerHub using the command below:

|  |
| --- |
| docker image push $DOCKERID/linux\_tweet\_app:1.0 |

1. Go to https://hub.docker.com/r/<username> and you should see your repository.

This is currently public but if you log in and explore the settings you can make it private if you wish to do so.

1. Push your second image to the Hub as well and make sure you can see both on the website.

Congratulations! You have completed the fourth lab in this module. Docker will be used extensively throughout this module so it would be a good idea to make sure you are familiar with its use before continuing.

**PLEASE REMEMBER TO STOP OR TERMINATE YOUR VM AT THE END OF THIS LAB SO THAT YOU DO NOT USE UNNECCESARY CREDIT!!**