CMPSC 100-03 Lab Session 1: Docker & Unix

- Assigned: 2 September 2019
- Due: 9 September 2019
- Point value: 20 pts

In this laboratory session, we concentrate our efforts on in-depth learning about how Docker & Unix (Ubuntu "flavor") work. We explore (in a partially particular order):

- Building Docker files
- Running Docker files
- Adventures in Unix
- Commands such as: `cd`,`ls`,`pwd`,`mv`,`rm`, `mkdir`
- Traversing directories
- Reading files
- ...& more!

General guidelines for laboratory sessions

- **Follow steps carefully.** Laboratory sessions often get a bit more complicated than their preceeding Practical sessions. Especially for early sessions which expose you to platforms with which you may not be familiar, take notes on `commands` you run and their corresponding effects/outputs. If you find yourself stuck on a step, let a TL or the professor know! Laboratory sessions do not mean that we won't help you in the same way we do during Practicals.
- **Regularly ask and answer questions.** Some of you may have more experience with the topics we're discussing than others. We can use this knowledge to our advantage. But, like in Practicals, let students try things for a while before offering help (**always *offer* first**). To ask questions, use our Slack (https://cmpsc100fall2019.slack.com) 's `#laboratory` channel.
- **Store and transfer files using GitHub.** Various forms of file storage are more or less volatile. *You* are responsible for backing up and storing files. If you're unsure of files which have been changed, you can always type `git status` in the terminal for your working folder to determine what you need to back up.
- **Keep all of your files.** See above, but also remember that you're responsible for the files you create.
- **Back up your files regularly.** See above (& above-above).
- **Review the Honor Code (https://sites.allegheny.edu/about/honor-code/) reguarly when working.** If you're taking a solution from another student or the Internet at-large (*especially Stack Overflow (https://stackoverflow.com) , bear in mind that using these solutions*

> ***_can*** constitute a form of plagiarism that violates the Allegheny Honor Code. While it may seem easy and convenient to use these sources, it is equally easy and convenient to rely on them and create bad habits which include not attributing credit or relying exclusively on others to solve issues. Neither are productive uses of your intellect. Really.

Further helpful reading for this assignment

If you have not already done so, I recommend reading the GitHub Guides (https://guides.github.com) which GitHub makes available. In particular, the guides:

- Mastering markdown (https://guides.github.com/features/mastering-markdown/)
- Documenting your projects on GitHub (https://guides.github.com/features/wikis/)
- GitHub Handbook (https://guides.github.com/introduction/git-handbook/)

In adddition, you may find a cheatsheet for Unix `commands` helpful throughout your time here. I recommend this one (https://files.fosswire.com/2007/08/fwunixref.pdf) from FOSSwire. You may not need all of these in this course, but someday I guarantee you'll have to do something obscure in Unix. This sheet, of course, is only a start to the many, many different strange (& often accidental) combinations that you can make with `commands` and "piping."

As for a markdown cheatsheet, this GitHub repository (https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet) serves as a useful guide.

If you find either helpful, I suggest bookmarking them.

Table of Contents

Your assigment

Broadly this sessions goals are:

- To learn how to build and run Docker `containers` from "`images`"
- To learn how to work with GitHub & GitHub Classroom
- To discover basic operations in the Unix operating system (OS)
- To help Prof. Luman find his cat

Beginning the assignment

Accepting the assignment

- [ ] Log into the `#labs` channel in our class Slack (https://cmpsc100fall2019.slack.com)
- [ ] Click the link provided for the lab assignment and accept it in GitHub classroom
- [ ] Once the assignment finishes building, click the link for your personal repository assignment

Cloning the repository

- [ ] On the resulting repository's page, click the `Clone or download` button
- [ ] In the upper right corner of the box that appears, click `Use SSH`
- [ ] Copy the link that appears in the textbox below the phrase "Use a password with a protected key."
- [ ] Open a terminal window using:
  - Your operating system's native terminal (Mac OS & Unix)
  - Git Bash (Windows)
    - Git Bash is located in the "Start" menu under the "Git" entry.
- [ ] Type `cd ~` to go to your home directory, or choose an easily-accessible, memorable location
- [ ] In the terminal window clone the repository
- If I (the instructor) were to clone my own repository, I'd enter:

```
git clone git@github.com:allegheny-college-cmpsc-100-fall-2019/cmpsc-fall-2019-lab-01-dluman
```

- [ ] Once finished, cd to the directory where the repository was cloned

A deeper dive into Docker

Docker relies on three (3) fundamental object types:

- `imageS`
- The `Dockerfile`
- `containerS`

In general terms the relationship between the three is described as: `Dockerfile` -> `image` -> `container`

`images`

This is an `image`:

But, it's not what we mean when we say `image` when we talk about Docker.

The term `image`, in this context, refers to a "snapshot" of a computer's ***secondary memory***. The complete picture of everything on your computer ***right now*** could constitute an `image`. In fact, we ***could*** build a `image` of your computer and install it on someone else's, effectively creating a copy of your entire `file system`.

Docker uses the term `image` to describe the product of a "snapshot" of a virtual system, and is created by a `Dockerfile`.

The `Dockerfile`

These files contain `commands` which help build an `image`. They can be simple or complex, depending on what a user needs a `container` to do. General practice for using `containers` is to pack "light," or only include the ***absolute minimum*** necessary to perform a specific operation or run a specific `program`. The following is an example of a `Dockerfile` that builds a simple Unix `container`:

```
FROM ubuntu
```

Here, `ubuntu` is what is referred to as a ***base `image`***, or the minimum possible configuration for starting a commonly-used resource--in this case, the Unix distribution named "Ubuntu" (the same Unix "flavor" that we will use in this course). We will use two `Dockerfiles` using this `image` later in this exercise.

You will not need to build any `images` from scratch in this course.

`containers`

A `container` is simply an `image` that is ***currently running***. This is the product of a Docker `command` (`run`) that we will also look at below. For now, the relationship between a Docker `image`, `Dockerfile`, and a `container` is all we need to know. However, I recommend exploring Docker functionality on your own time to enhance your understanding.

Running a `container`

In this part of the exercise, we will run the Ubuntu ***base `image`***.

Docker Desktop users

- [ ] Open a terminal window

Docker Toolbox users

- [ ] Open the "Docker Quickstart Terminal"
- When this lab assignment uses the word `terminal`, it is interchangeable with the Docker Quickstart Terminal

Base `images`

- [ ] In your terminal window, type `docker image`
- Make a note of what you see; what does this list display?
- Are there any items there? Where did they come from (think back to the last Practical)?

We need to acquire the "Ubuntu" ***base*** *`image`* so that we can begin our work. We will use the `pull` `command`.

- [ ] In the terminal, type `docker pull ubuntu` and press `Enter`

Docker will always look ***locally*** (that is, on your computer) for `images` first. If it does not find any with the name specified (here `ubuntu`), it will look at a common location known as "Docker Hub" (in the cloud) and download the matching name. You will see something like the following:

```
Using default tag: latest
latest: Pulling from library/ubuntu
Digest: sha256:d1d454df0f579c6be4d8161d227462d69e163a8ff9d20a847533989cf0c94d90
Status: Downloaded newer image for ubuntu:latest
```

- [ ] Type `docker image` again and press `Enter`
- What appears in the terminal now?

These `images` are available to you wherever you're working on your system and can use Docker.

Making a `container`

To turn the `ubuntu` `image` into a `container`, we need to use the `docker run` `command`. But, before you do, we need to review a bit about how `commands` work.

`commands` & flags

In this assignment, we've already used some `commands` and `programs`.

- [ ] Which have been `programs` and which have been `commands`?
- Hint: `docker` is a `program`.

Many `commands` feature options referred to as `flags`. These are usually prefixed by a single dash (`-`) or double dash (`--`). `-` `flags` generally represent "shortcuts," and `--` `flags` are the direct, generally longer, names. Not all `flags` have both `-` and `--` prefix versions, though. This is a `program`-by-`program` preference/issue.

For future reference and help, both `programs` and `commands` generally contain a `--help` or `-help` `flag`. Try it now:

- [ ] In the terminal, type `docker --help`.
- What does this list (are they `programs`, `commands`, or `flags`)?
- [ ] In the terminal type `docker run --help`.
- What appears here (`programs`, `commands`, or `flags`)?

We will revisit this conversation as we explore Unix further today.

The `container`

- [ ] In a your terminal window, type `docker run -it ubuntu` and press `Enter`
- What are `-i` and `-t` (`programs`, `commands`, or `flags`)?
  - `-i` initiates what is called an "interactive" session (using something called `STDIN` to allow you to type in the `container`)
  - `-t` begins a TTY (literally, "**t**alk **t**o **y**ourself) session which displays what the operating system "says" back to you (something called `STDOUT`)
  - When supplying two (2) `flags` to a `command`, generally we can combine them unless we have more specific instructions for each `flag`.
- What happened when the `command` & `flags` executed? What displays in the screen?
- [ ] Type `docker`images `in the terminal window and press`Enter`
- What happens? Why?
  - Hint: are you still interacting with ***your*** computer or something else?

First foray into Unix

As if magic (but it really is magic), you have been transported into a Unix operating system session on your computer which ***runs a completely different operating system***. Think of this like the "picture-in-picture" feature on a TV; yes, you can watch two "channels" at once. You are inside a running `container` which "boots" directly to a `command` prompt.

Unlike a terminal session, which may allow some of the same `commands`, this is a fully-fledged Unix system. However, thinking back to the brief discussion of the purpose of `images` and `containers` above, it's ***extremely minimal***. Not everything is available. But, let's explore what we can do.

The Unix prompt

You should see something similar to the following:

```
root@0659308ad943:/#
```

This is the Unix `command` prompt in "bash" format. This format display the user name, an `@` symbol, the machine's "host name," followed by colon and the `current working directory` in which the operating system is `mounted`, or accessed.

- What is the user?

- What is the host name?

First I must caution you, young Jedi: with great power, great responsibility.

In this `container` you have assumed the magical guise of a user known as `root`. This is a very powerful position. Within reason, as `root`, you can do ***anything you want*** in a computer system. Be careful, because here you will get the result which match exactly what you asked the system to do, whether it was want you wanted or not. A reminder of this power is that the character which ends the `command` prompt is an octothorpe `#` (a.k.a. "pound" or "hash" sign).

Basic `commands`

Traversing a `file system`

- [ ] At the `command` prompt, type `ls` (short for **l**i**s**t) and press `Enter`
- This `command` displays a listing of files in the current "working directory."
- [ ] To find the "current working directory," type `pwd` and press `Enter`
- This `command`'s syntax stands for "**p**ath to **w**orking **d**irectory."
- What displays?

`File systems` in Unix begin with `/` characters. The `/` path represents that this is the ***lowest*** level. From here, we can access various other `file systems`. Some here include:

- `/bin`: The place where basic `binaries` (Unix `programs` & `commands`) live
- `/root`: The `root` user's home space
- `/home`: The directory where other system users can find their home space
- `/var`: A `file system` which often contains log files for various `programs` and `processes`
- & others

It is not necessary to memorize these for this class except to know that most of the time, you'll want to be working in the `/home` file system.

- To explore the `/bin` `file system`, we need a mechanism for changing directories.
    - If you had to guess, what a short `command` might we use to **c**hange **d**irectories?
- [ ] Type `cd bin` at the `command` prompt and press `Enter`
- How does the `command` prompt change?
- What does this change reflect?
- [ ] Type `pwd`
- What displays?
- Is this what you expected?
- [ ] Type `ls`

- What displays?

Here, you can see a list of all of the `commands` and basic `programs` available in this `image` of the operating system (Ubuntu). In many systems, this directory is quite a bit larger. Here, because the basic principle of `containers` is to package only what is ***necessary*** only the most commonly used functions are available.

There are a number of shortcuts that one can use to "traverse" (or, move around) a `file system`. For example, let's return to the previous level of the `file system`. To do this:

- [ ] Type `cd ..` and press `Enter`
- What did this `command` do?
- The `..` (the value we entered) is referred to as a `parameter` or `value`

We can go forward only if we know exactly where we want to go:

- [ ] Type `cd /var/log` and press `Enter`
- We have traversed the `var file system` to a depth of 2, here.

However we can go back if we either know:

- The exact location we want to go to, or
- The relative depth of where we are
- [ ] Type `cd ../../` and press `Enter`
- [ ] Type `pwd` and press `Enter`
- Where are you now?
- What does `../../` represent?

Creating files

- [ ] Type `ls /bin` and press `Enter`
- What did this `command` do?

In the list generated by this `ls command`, you should see a `program` called `touch`. This `program` creates files without opening them. Let's create a file in our `root` user's home directory, `/root`.

- [ ] Type the `command` required to get there.
- If you're stuck here, ask a neighbor and then a TL or the professor!
- [ ] Once in the `root file system`, type `ls` and press `Enter`
- What appears?
- Why might the result of this `command` be what it is?
- [ ] Type `touch testfile` and press `Enter`
- [ ] Type `ls` and press `Enter`
- What appears now?
- Did the output change? Why?

- [ ] Type `touch file1 file2 file3 file4` and press `Enter`
- [ ] Type `ls` and press `Enter`
- What was the result of the last two `commands`?

Making directories

To make a directory in Unix, use the `mkdir command`.

- [ ] Type `mkdir testdir` and press `Enter`

You should be able to traverse to this directory now.

- [ ] Enter the `commmand` to change to this new directory.
- How might you do it?
- How do you know you're in the directory?

The `mv command`

Renaming in Unix may seem a little less straight-forward than other operating systems (OS). However other OSs merely orchestrate the same operation that Unix does to make the `process` appear differently. Essentially, when you're renaming a file in an OS, you're moving it from one memory location to another.

- [ ] Type `mv testfile testfile.old` and press `Enter`
- [ ] Type `ls` and press `Enter`
- What appears?
- Note: the `cp`, or **c**o**p**y `command`, works the same way! Instead of moving files, it makes a copy.

The `rm command`

To delete a file, Unix uses the `command rm`.

- [ ] Traverse back to the `/root` directory.
- [ ] Type `rm testfile.old` and press `Enter`
- [ ] List the contents of the directory
- What happened to your file?
- [ ] Type `rm testdir` and press `Enter`
- What happened?

Directories require users to add an additional piece to the `command` to remove directories.

- What is that piece called?
- [ ] Type the correct `command` to remove the directory
- [ ] List the directory again

- Is your directory gone?

Exiting a `container`

To exit a `container`:

- [ ] Type `exit` at the `command` prompt and press `Enter`
- Where are you now?
- How do you know?

Working inside a `container`

Now, let's return to our `container` and work on `file1` – `file4`

- [ ] Re-run the previous `container`
- See above for the `command`, or, if stuck, as a neighbor, TL, or the professor
- [ ] Traverse back to the `/root` file system.
- [ ] List the contents of the directory.

Uh...where's your work?

Oh, you lost it. Good job.

I don't accept the "`container` ate my homework" excuse. Nice try.

Actually, that was **my** fault. I didn't explain a key concept about `containers`: they're what we refer to as ***transient***. Think back to the previous discussion the class had about ***main memory*** and its relationship to `containers`.

- How does this explain what just happened?
- Call a TL or the professor over and discuss your thoughts. Take notes on your discussion.
- [ ] `exit` the `container`

Second foray into Unix

For this adventure, we're going to access a `Dockerfile` written specifically for this class session. If you were to open the file in your text editor, you'd see quite a few different instructions written so that Docker can build a more specific `image`.

In your computer's terminal (be careful that you're not still in a `container`, if so: `exit`):

- [ ] Ensure that you're back in the main directory where you cloned the repository
- Protip: in Unix and Unix-like terminals, if you write **part** of the title of the folder or file

you'd like to access, hitting the `Tab` key often completes it at the prompt. Try it. If you have difficulties call a TL or the professor over to demonstrate.

- [ ] Change directories to the `1-ubuntu-explore` folder.
- [ ] In this folder, type `docker build . -t ubuntu-explore`
- This `command` uses a `parameter` and a `flag`.
    - The `-t` flag for the `build command` is different than that of the `run command`. Here, `-t` means "tag," which labels the `image` for easy access.
    - The `.` tells Docker to build the `Dockerfile` at the current working directory's "absolute" path. To find the absolute path type `pwd` at your terminal and press `Enter`.

Building another `image`

Once the `process` completes, you should have a new `image` titled ("tagged") "ubuntu-explore."

We're going to add another `flag` to this Docker instance to avoid the issue we just had when we attempted to access a `container` and found all of our files gone.

- [ ] At your terminal prompt, type:

```
docker run -it --mount type=bind,source="$(pwd)",target=/home/cmpsc100/mount
 --hostname ubuntu-explore ubuntu-explore
```

Here we added two `flags` with several `parameters`. Let's break them down:

`--mount`

Like any computer, users have to attach ***secondary memory*** to store files in a semi-permanent manner. The last example disposed of all of the files because `containers` load into ***main memory*** only. Here, we're attaching (`mounting`) our hard drive's current working directory to the `container` so that we're saving our work.

The `parameters` we've added include:

- `type=bind`
    - This creates a "bi-directional" connection to your local computer's ***secondary memory***. This is a `read-write` connection, meaning that any changes we make in the `container` are written and saved to **your** machine. *`source=$(pwd)`
    - This is a bit of a magic trick. Here, we use `$(...)` to leverage the output of the `pwd command` so that we don't have to worry about knowing ***exactly*** where we are. This is the place that files written while in the `container` will end up.
    - Protip: the `$(...)` structure works for a ton of `commands`. Looking up how others use it will open up a world of Unix `command` prompt wizardry to you. *`target=/home/cmpsc100`
    - Again: MAGIC! As part of the `Dockerfile` I created, there's a `/home/` directory where we'd like to work with and save items. This is the place ***in the*** *`container`* where any files in the current working directory `$(pwd)`

`--hostname`

To avoid the word salad that is a general `container`'s name (I for one don't know what `f9f90b2f4cf1` means, for example) we can provide a name to the `container` at what we refer to as "runtime." Here, we've chosen `ubuntu-explore`, but it really could be ***anything***.

- Note: for the remainder of this lab, the document will refer to the general ideas of `commands` rather than the `commands` themselves. If introducing new content, it will be described and specified.

Working with mounted drives

You should now be in your brand new, shiny `container`.

- What user are you now?
    - How do you know?
- Did you start in a different directory?
    - How do you know that, too?
- What else is different about the `command` prompt?
- [ ] Determine where you are, and what the directory's contents are.
- It looks like there's something there, and it's very important...let's inspect.
- [ ] Traverse to the new directory
- [ ] List the directory's contents

Let's peek into some files to see what they contain.

- [ ] Type `less all.csv`
- `less` is a quick way to view the contents of a file.
    - To scroll up and down, use the `Page Up` and `Page Down` keys.
    - To exit `less` hit the `q` key.
    - If you get stuck in `less` (which is possible) let a TL or the professor know.

It looks like someone has shipped us some very important data in this `container`, and we want to save it. Thankfully, ***we mounted our drive, and can move it all to a safe place***!

But! The Gator Wizard is upset. He sees that there are CSV files in this directory and thinks that the extension stands for "**C**rocodile **S**ave **V**ersion!" He wants us to cast a spell to **delete** them!

Thankfully, we have a handy trick to do that.

- [ ] In the terminal window, type `rm *.csv`
- This leverages the `*` character, a "wildcard" which allows us to specify deletion for any

file with the extension `*.csv`

- [ ] To move all of the files to our mounted drive, in the terminal type `mv *.txt ../mount`
- This `command`:
    - Traverses one directory level **lower**
    - Enters and moves files to the `mount` directory--the directory connected to our local drive
- For those looking for a shortcut, the ~ is useful convetion in Unix. It represents the user's `/home/` directory.
    - At any time, you can `cd ~` to return to `/home/cmpsc100` or
    - `mv` items to ~/mount

To test this (**on your local machine, leaving your `container` running in the terminal**):

- [ ] Open the `1-ubuntu-explore` folder
- [ ] Check that the contents are there

Test **in the `container`**:

- [ ] Traverse to the `mount` directory
- [ ] List the directory's contents.

Once you've verified that the transaction has taken place:

- [ ] `exit` the `container`

Meowtown: a Docker Game

This game involves everything you've learned so far in this lab exercise, which includes:

- Building `image`s
- Running `container`s
- Basic Unix skills
- Using Unix `command`s to:
    - Traverse `file systems`
    - Manipulate files

The goal of this game is to find Prof. Luman's cat wherever it lives in your `container`'s `file system`! This is randomly generated on each build, so no two users' experiences will be the same.

- At any time, type `rules` in the terminal to display the message given at container start-up

Back on your local machine:

- [ ] Traverse to the `2-meowtown` directory.

- [ ] `build` the Docker `image`, and tag it `meowtown`
- [ ] `run` the `container`:
- `bind`ing your local machine's drive current working directory to the `/home/cmpsc100/mount` directory
- Setting the `hostname` to "meowtown"
- Refer to the sections on mounting, bind, source, target, and hostname above.
- As always, if you're stuck, ask a neighbor, TL, or the professor!

Rules

This game involves moving through directories and copying or moving or copying files.

- There are several pictures of my cat, Ulysses, scattered among the various directories in the `image`.
- Your job is to find them all and copy them to the `/home/cmpsc100/mount` or `~/mount` (samed place) directory.
- When you are finished, type `done` at the `command` prompt to finish the game.
- Hints:
    - In the `container`, type `answers` at the `command` prompt to check how many pictures of Ulysses there are.
    - At any time, typing `done` at the `command` prompt will update you on your progress.

When finished:

- Where did all of the files end up?
    - You may need to look through all of the directories in the repository.

Finishing & reflecting on the activity

Now that you have completed the first lab, there are two items remaining on your checklist to recieve credit (using your local machine, not a `container`):

- To write a reflection on your experience (using Atom (https://atom.io) , or another text editor)
- To `commit` your working repository to GitHub

`Write` a reflection

- [ ] In the main folder of the repository, create a file named `reflection.md`
- [ ] Using markdown (see resources above), write a reflection on difficulties, successes, joys, frustrations of the lab experience in addition to answering:
    - Given your experience with this assignment, how would you describe the relationship of `Dockerfiles`, `containers`, and `images` in your own words?

- About what points do you still have questions?

`Commit` your repository

For the following tasks, you will need to use a terminal.

- [ ] In the main folder of the repository (`cmpsc-100-fall-2019-lab-01`) type:

```
git add .
```

- This `command` searches the directory for anything you've changed or added and "stages" them for transmission to GitHub.
- [ ] Then, type:

```
git commit -m "{DESCRIPTIVE COMMIT MESSAGE HERE}"
```

- This adds a message to your commit detailing what you've change ***or*** the purpose of the commit (e.g. "Turning in the assignment.")
- [ ] Finally, type:

```
git push origin master
```

- This will send all of your files to the GitHub repository for your version of the assignment.

After the commit, the additional files should appear on the page for your repository.