SENG265 Lab 2

This lab explores the version control system called git. Throughout this lab, input and output will include the characters `NETLINKID` - when this appears, substitute your own netlink id instead. Note that this may be appear multiple times in a single command.

By the end of this lab, you will be able to: - clone your git repository from your *master* repository - modify the files in the resulting *local* repository - use shell redirection to send output to a file - add modified files to the set of files tracked by git - write a commit message explaining your changes - push the *local* repository back to the master repo - confirm that your changes have been saved by the master repo

Repositories

Your files in this course reside in *git repositories*. There is a central, or *master* repository stored in a data centre somewhere on campus.

The files in this repository are the ones we will mark. However, you are not able to edit this *master* repository directly. Instead, you must create a copy - a *local repository* on the computer you're working on. You modify the files in this *local repository*, and when you're done, you *push* this local repository back to the master in the data centre. This lab will give you a chance to try out this workflow.

Begin in your home directory

Remember from lab one that your terminal, by default, opens with the working directory set to your home directory (also known as ~) The command prompt will look something like this:

```
[NETLINKID@ugls10 ~]
```

You will, of course, see your own username instead of `NETLINKID`, and will probably have a different computer number than that shown above. If you are not in your home directory, you can return there with `cd ~`

Introduce yourself to git

Before we can use git, we need to configure it by telling it who we are. Execute the following commands to tell git your name and your email address. You only need to do this once; git will remember.

```
[NETLINKID@ugls10 ~] git config --global user.name "Firstname Lastname"
```

You should replace 'Firstname' and 'lastname' with your own first and last names.

```
[NETLINKID@ugls10 ~] git config --global user.email "NETLINKID@uvic.ca"
```

Again, replace `NETLINKID` with your own netlink id.

Confirm that it worked:

```
[NETLINKID@ugls10 ~] git config --list
```

Clone your repository

Last week we created a directory for this course; go into that directory now with the `cd` command. (If you don't have a directory for the course, remember you can create a directory with the `mkdir` command).

To create a local copy of your master repository (what we call a *local repository*) use the git clone command:

```
[NETLINKID@ugls10 ~]$ git clone ssh://NETLINKID@git.seng.uvic.ca/seng265/NETLINKID
```

In both cases, of course, you'll replace `NETLINKID` with your own netlink id. After authenticating to the system, you'll have a new directory available - in this course, it is named after your netlink id (although repositories can also have other, more descriptive names).

Explore your repository

Change to this new directory and look around. You have four directories named `a1` to `a4` - these will be for your four assignments. You'll place your assignment code in these directories, and we'll mark whatever we find there.

There's also a directory called cheatsheets. Most of these you'll need to view in the GUI, but bash-cheatsheet.sh is just text, and you can view it with the `cat` command.

If you go back to the 'root' directory of your repository (the directory called NETLINKID), you can use the `ls -a` command to view hidden files and directories (ones with a period at the start of their name are hidden by default). Once of these is the `.git` directory. Try changing to the `.git` directory and take a look around inside.

These files and directories are keeping track of any changes you're making to the repository. When it comes time to *push* this *local* repository back to the *master* in the data centre, these files and directories will tell git how to modify the *master repository*.

Finding out the status of the repository

Go back to the root of the repository, and issue the `git status` command. You'll see:

```
# On branch master
nothing to commit, working directory clean
```

This means that you have not yet made any changes to your local repository - it's identical to the master in the data centre. Try making a change: create a Lab2 folder with the `mkdir` command.

Now type `git status`, and you'll see the same message:

```
# On branch master
nothing to commit, working directory clean
```

What's going on?? You made a change (you created a directory), but git hasn't noticed your change - why? This is because git tracks *files*, not *directories* - empty directories don't exists, as far as git is concerned. Let's fix that.

Go into your `Lab2` folder and create an empty file, named `commit_log.txt` with the `touch` command.

Now go back to the root of your repository and try `git status` again.

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   Lab2/
nothing added to commit but untracked files present (use "git add" to track)
```

Git now knows that something has changed in `Lab2`, but it hasn't been told to *track* those files - to monitor them and keep track of changes made to them. We can tell git that we want it to start tracking files in `Lab2` with the `git add` command (as the status message suggests):

```
[NETLINKID@ugls10 NETLINKID]$ git add Lab2
```

If we check the status again, we see:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   Lab2/commit_log.txt
#
```

Git now knows about our new file, and even better, the red text has changed to green.

Write a commit message

We've made a change, but we haven't explained WHY we've made the change. If we ever need to revert to some previous version of our code, we want to make bookmarks in the revision history of our code - particular points in the past that have useful messages that explain what we were doing. These are called *commits*, and we create them with the `git commit` command:

```
[NETLINKID@ugls10 NETLINKID]$ git commit -m "adding the commit_log.txt file"
```

If you don't add the -m and a message in quotes, git will start up `vim` to allow you to type a more complete message. If `vim` is killed, no commit will happen - in general for short messages it's easier to just use -m.

Let's check the *commit log* - the history of all the commits that have ever been made in our repository. We do this with the `git log` command. Here's what mind looks like; yours will have different dates and times, but will be similar.

"'commit dc641d734dece1abe0f3c572b27fd66125f53476 Author: David Clark daclark@uvic.ca Date: Thu Sep 15 13:22:33 2022 -0700

```
adding the commit_log.txt file
```

commit a5872155b1f67289e299d0a236ea79a9bc85c819 Author: Lynn Palmer lpalmer@uvic.ca Date: Fri Sep 9 11:29:01 2022 -0700

```
initial


There are two commits in this log - the first one is at the bottom, and shows the repo being

Redirection
--

Right now our `Lab2/commit_log.txt` file is empty - let's put the output of the above `git l
```

[NETLINKID@ugls10 NETLINKID]$ git log > Lab2/commit_log.txt "'

Use the `cat` command to confirm that the above log text really did end up in `Lab2/commit_log.txt`

Make git aware of our new change

We're ready to send our changes back to the *master* repository in the data centre, but first we need to make sure that git is aware of our new changes. `git status` tells us what's going on:

```
# On branch master # Your branch is ahead of 'origin/master' by
1 commit. #   (use "git push" to publish your local commits) #
# Changes not staged for commit: #   (use "git add <file>..."
to update what will be committed) #   (use "git checkout --
<file>..." to discard changes in working directory) # #   modified:
Lab2/commit_log.txt # no changes added to commit (use "git add"
and/or "git commit -a")
```

This message says that git knows about our first commit, but that we have changes that it doesn't know about (they're not 'staged for commit'). We can fix that by writing another commit message, this time using the `-am` option to add the updated version of the file at the same time:

```
[NETLINKID@ugls10 NETLINKID]$ git commit -am "commit log has text in it now"
```

Now if we do `git status`, we see that we are '2 commits ahead of *master*', which makes sense - we've done two `git commit` commands since we cloned the repo.

Push your changes to the master repository

We're nearly done. We've made changes to our local repository, but the *master* repository in the data centre hasn't changed. We need to *push* our repository to the master, with the `git push` command. When we `git push`, we are interacting with the data centre, so just like when we did a `git clone`, we may be asked for our UVic credentials.

```
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (8/8), 828 bytes | 0 bytes/s, done.
Total 8 (delta 2), reused 0 (delta 0)
To ssh://NETLINKID@git.seng.uvic.ca/seng265/NETLINKID
   a587215..afc45f0  master -> master
```

The master repository now has our changes.

How can you be sure?

The best way to confirm this is to try cloning the repository again! We can have as many local repositories as we want. Create a new one by going back to your home directory and creating a new directory there. Change to your new directory, and use the `git clone` command to clone a new remote repository. Enter that new repository, and if you have a `Lab2` directory with a file inside named `commit_log.txt`, then you know you successfully updated your master repository. If not, go back to the first copy of your repo and use the `git status` and `git log` commands to find out where things went wrong.

Submit a screenshot

Take a few minutes to take a picture of your git commit log. Put the log up on your terminal with the `git log` command, and take a screenshot using the KSnapshot application built in to the desktop (you can find it from the menu in the upper left corner of the screen at `Applications -> Graphics -> KSnapshot`). Save it on your drive, and submit it to the `Lab 2 Screenshot submission` assignment, under `Course Tools -> Assignments`.

About `git pull`

You don't need to clone your repo every time you want to work on it. If you already have a *remote repository* around, you can enter it and use the `git pull`

command to update it. This is useful if you work at several different computers.

Let's say you're working on your desktop. You've done a `git clone` and created a *remote repository* (a copy of your master repository) on your desktop.

You then use your laptop, and do a `git clone` there. Now you have two remote repositories, one on each machine. You make some changes on your laptop's repository, and push them back to the *master* repo in the data centre.

Now your laptop and the data centre have the new updated versions of your files, but your desktop is behind - it has the old versions. To update your desktop's repo to bring it up to speed with the others, just do a `git pull` command from within your desktop's repo. Your desktop will connect with the *master* repo in the data centre, and will update itself. Now you're working with the latest version of your files!

Want more practice?

We created a Lab2 directory today, added it to our repo, and pushed our changes to the *master* repository. Last week, we worked with some files in a lab-01 directory. Try adding them to your remote repo - you'll need the `cp` command, and remember that `cp -R` copies from the source to the destination *recursively* - that is, it copies files, directories, and the contents of directories. Once you're finished, do `git add` and `git commit` in order to make git track these changes, and push your changes back to the *master* repo. Finally, try cloning another copy of your repo to confirm that your changes have been committed correctly.