Git Lab 2
Tracking changes and settings

1 Commands covered

Examining Git objects
`git cat-file -p` *`object-SHA1-hash`* `git cat-file -t` *`object-SHA1-hash`* `git show` *`object-SHA1-hash`* `git show --oneline` *`object-SHA1-hash`* GitHowTo Git internals and objects Atlassian git show tutorial

Saving changes	**Tracking changes**

`git add --all`	`git log`
	`git log --patch`
	`git log -stat`
	`git log` *`commit-hash`*
	`git log` *`commit-hash1..commit-hash2`*
	`git log -n` *`x`*
	`git log --pretty=oneline`
	`git log --oneline`
	
	FreeCodeCamp git log command
	
	Atlassian git log command

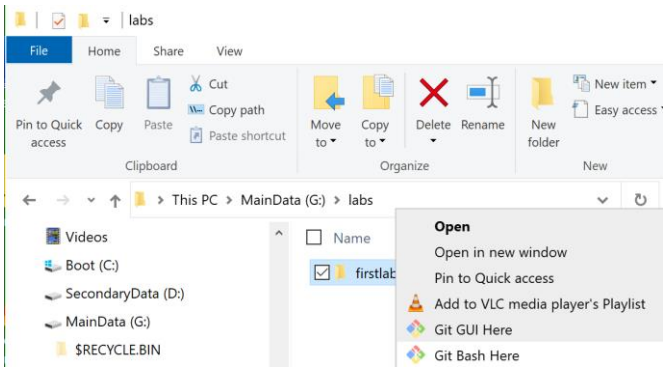
	`git diff`
	`git diff --staged`
	`git diff commit-hash-A..commit-hash-B`
	
	Atlassian git diff command
	
	git diff complete tutorial

2 Lab setup

Make sure you have a suitable text editor installed for your OS.

We will continue with the Git project from the previous Lab 1 in the working directory `labs\firstlab`. If are using a directory with a different name, substitute that name into all the commands that use this directory name in this lab session.

If you are working with Linux / MacOS, open a terminal shell to type in your Git commands.
If you are working with Windows, we will be typing in the commands via Git Bash. Right click on the directory and select Git Bash here from the context menu to open the Bash shell.



The Git commands to type are listed after the $ prompt and their output (or relevant parts of their output) will be shown as well.

3 Examining Git Objects

The Git objects that are core to the Git architecture are stored in the Git repository in the `.git` folder in the project root folder. If you navigate into this folder using the file manager utility of your OS, you will be able to see a variety of folders and files.

The `objects` folder stores the various Git objects in folders with names of 2 characters which correspond to the first 2 characters in the SHA1 hash for that object. These folders typically contain a single file whose name correspond to the remaining characters (after the first 2 characters) of the SHA1 hash for that object.

Perform a `git log` and identify any commit, then use its hash to identify the corresponding folder and file holding its contents in the `objects` folder. Open this file in your text editor. Notice that it contains gibberish: this is a compressed form of the actual content of that commit.
CAUTION: Do not attempt to modify this content as you risk corrupting the repository !

Select the same commit that you used earlier, and then type this command below to view its contents. When specifying *`object-SHA1-hash`*, you do not need to type all the 40 characters of the SHA1-hash, just enough characters (usually between 5 - 7 is adequate) to uniquely identify it from all other objects available in the repository.

```
$ git cat-file -p object-SHA1-hash

tree 08f7b4cd841812e9888ac76afc3d12ecfb7650a5
parent 7500f29d4373002ed320f5f2bb6e231f171ea636
author Peter Parker <spiderman@gmail.com> 1646040440 +0800
committer Peter Parker <spiderman@gmail.com> 1646040440 +0800

Renamed people to humans and added 3rd line to all files
```

In addition to commit metadata such as author, committer and commit message, you can see the SHA1-hash for the tree object that the commit references as well as its parent commit.

We can check the type of the object with the same command but a different option.

```
$ git cat-file -t  object-SHA1-hash
commit
```

We can examine the content and verify the type of a tree object in a similar manner:

```
$ git cat-file -p object-SHA1-hash
100644 blob 12ba8b00e2112f1164586b0e243c36cc85df4bf5    animals.txt
100644 blob 7dddefec5efc9563e22ca3042698c4d759ba25bc    cars.txt
100644 blob 76a19d10dd663ab5d50dbf0321f885258ca5e08e    humans.txt

$ git cat-file -t  object-SHA1-hash
tree
```

We can examine the content and verify the type of a blob in a similar manner. Notice now that the contents of the blob are in their actual form.

```
$ git cat-file -p object-SHA1-hash
1: cat
2: dog
3: elephant

$ git cat-file -t  object-SHA1-hash
blob
```

The `git show` command provides more detailed information on a given object (commit, tree, blob or tag). For e.g. when used on a commit, we have:

```
$ git show commit-SHA1-hash
```

```
commit 3cace89fd3a05b998f9223c8154c66486ccc3076 (HEAD -> master)
Author: Peter Parker <spiderman@gmail.com>
Date:   Mon Feb 28 17:27:20 2022 +0800

    Renamed people to humans and added 3rd line to all files

diff --git a/animals.txt b/animals.txt
index a791d70..12ba8b0 100644
--- a/animals.txt
+++ b/animals.txt
@@ -1,2 +1,4 @@
….
….
….
```

This provides more information on the commit such as the branch that is pointing to it, as well as the difference in contents between this commit and its immediate parent commit. We will be looking at using the diff command to show content difference in an upcoming lab.

We can also use git show on the hash for a tree and blob object as well:

```
$ git show tree-SHA1-hash
tree 08f7b

animals.txt
cars.txt
humans.txt

$ git show blob-SHA1-hash
1: cat
2: dog
3: elephant
```

There are several options available for the `git show` command to summarize its output. Let's try out one of these options on a commit object.

```
$ git show --oneline commit-SHA1-hash

3cace89 (HEAD -> master) Renamed people to humans and added 3rd line
to all files
diff --git a/animals.txt b/animals.txt
index a791d70..12ba8b0 100644
…..
```

If we return to the `.git` folder, you will see a `config` file. This contains the configuration variables at the local level, which means it applies only to this particular repository.

It also contains a HEAD file. This file specifies the branch name that the HEAD pointer is pointing at. In our example, it should have this current value:

```
ref: refs/heads/master
```

If we now navigate into that particular subdirectory and open the `master` file, we will see that it contains the hash of the commit that the MASTER branch is pointing to.

There is a file called COMMIT_EDITMSG which should now contain the last commit message that we created. If we do not specify a message at the time when we run the `git commit` command, Git will open this file in our default editor and prompt us to supply a message. We will be seeing this in action in a later lab.

4 Performing diff comparisons

In general, the modifications that occur within a file can be a combination of any of these 3 types:

i. Addition of new material
ii. Deletion of existing material
iii. Modification of existing material

Let's make more modification of these 3 types to the 3 files we have in the root folder and save:

Add the following two lines to the end of humans.txt

```
4: sales rep
5: HR manager
```

humans.txt

Remove the last two lines from animals.txt. It should now look like this:

```
1: cat
```

animals.txt

Alter the contents of cars.txt so that its new content looks like this

```
1: mazda
2: mercedes
3: perodua
```

cars.txt

We verify that all these changes are unstaged in the usual way:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
        modified:   animals.txt
        modified:   cars.txt
        modified:   humans.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Often, we want to check the differences between the current contents of our working directory after we have made some modifications with its state prior to those modifications. The command to perform a difference comparison between all unstaged changes in the working directory and their previous contents at the most recent commit pointed to by HEAD:

```
$ git diff

diff --git a/animals.txt b/animals.txt
index 40eb8d8..122d910 100644
--- a/animals.txt
+++ b/animals.txt
@@ -1,3 +1 @@
 1: cat
-2: dog
-3: elephant
…….
……..
……..
```

A long comparison output is shown for each of these 3 modified files. We will go through each of these comparison outputs one by one.

The first line:
`diff --git a/animals.txt b/animals.txt`
identifies the files being compared. In this case, these are two different versions of the same file. The term a is used to refer to the "earlier" version (i.e. contents at the most recent commit pointed to by HEAD) while b is used to refer to the "later" version (i.e. the new changes that are still unstaged).

The second line:
`index 40eb8d8..122d910 100644`
refers to the SHA-1 hash of the two versions being compared. The last number `100644` is the standard Git file permissions sequence that identifies the file as being read only.

The next two lines:
```
--- a/animals.txt
+++ b/animals.txt
```
identifies marker symbol (-) to be used for a (the "earlier" version) and (+) for b (the "later" version)

The next line is a chunk header
`@@ -1,3 +1 @@`
A chunk is basically the portion that has changed between the two versions. Git will show chunks instead of the entire contents of the file.

Lets look at the first part: `-1,3`
The – indicates we are referring to a (the "earlier" version). We are showing 3 lines starting from line 1.
Lets look at the second part: `+1`
The + indicates we are referring to b (the "later" version). We are only showing line 1.

Finally, lets look at the comparison itself

```
1: cat
-2: dog
-3: elephant
```

The first line is in grey and has no marker symbol in front of it. This indicates that it is common to both versions.

The next 2 lines are in red and have the marker symbol for a (the "earlier" version) in front of it. Red indicates that the lines were present in a but have been removed in b (the "later" version).

Lets scroll down or up to look at the next diff output:

```
diff --git a/humans.txt b/humans.txt
index 7f77bbf..9b3fd87 100644
--- a/humans.txt
+++ b/humans.txt
@@ -1,3 +1,5 @@
 1: developer
 2: project manager
 3: CEO
+4: sales rep
+5: HR manager
```

The first couple of lines are pretty much the same as before. The chunk header:
`@@ -1,3 +1,5 @@`
indicates we are showing 3 lines starting from line 1 from a (the "earlier" version) and we are showing 5 lines starting from line 1 from b (the "later" version).

Lets look at the comparison itself. The first 3 lines are in grey and do not have any marker symbols preceding them.

```
1: developer
2: project manager
3: CEO
```

This means these 3 lines are present in both a and b. The last two lines:

```
+4: sales rep
+5: HR manager
```

are in green and have the marker symbol for b, indicating that they are only present in the "later" version

Lets scroll down or up to look at the final diff output:

```
diff --git a/cars.txt b/cars.txt
index 9396493..bfa8716 100644
--- a/cars.txt
+++ b/cars.txt
@@ -1,3 +1,3 @@
-1: honda
+1: mazda
 2: mercedes
-3: proton
+3: perodua
```

Lets look at the actual comparison itself. The first two lines:

```
-1: honda
```

```
+1: mazda
```

indicate that both a and b have different contents for their first lines (red and the symbol – is for a, and green and the symbol + is for b).

The next line is in grey and has no marker symbol preceding it:

```
2: mercedes
```

We have already seen indicates that this line is present in both a and b.

The final two lines:

```
-3: proton
+3: perodua
```

again indicate that both a and b have different contents for their 3rd lines.

You may occasionally see the statement
`\ No newline at end of file`
in your diff output, depending on whether you actually included a newline (pressed the enter key) at the end of that specific file.

Note that the line numbers that we see in the diff output, for e.g:

```
@@ -1,3 +1,3 @@
-1: honda
+1: mazda
 2: mercedes
-3: proton
+3: perodua
```

are not placed there by Git, it is merely the actual contents of the file. In a typical file content where no line numbers are artificially placed there like was done in this example, the diff output would look like this:

```
@@ -1,3 +1,3 @@
-honda
+mazda
mercedes
-proton
+perodua
```

Lets stage all these changes that we have made. We can provide an option to `git add` that will stage all changed content in tracked files in the working directory and its subdirectories

```
$ git add --all

$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   animals.txt
```

```
        modified:   cars.txt
        modified:   humans.txt
```

Now if we run the previous command to perform a diff:

```
$ git diff
```

We get no output as there is no longer unstaged changes at this point of time.

If we would like to see the difference between the contents in the previous commit and all the staged changes, we would run:

```
$ git diff --staged
```

Whereupon we would obtain the same output as we did earlier.

Sometimes, when we have made a large amount of changes, we might only want to see the diff comparison for a specific file, rather than all the files involved in the changes. We can do that with:

```
git diff file-specifier
git diff --staged file-specifier
```

So, for example, we only wanted to see the differences for animals.txt, we would do:

```
$ git diff --staged animals.txt

diff --git a/animals.txt b/animals.txt
index 12ba8b0..122d910 100644
--- a/animals.txt
+++ b/animals.txt
@@ -1,3 +1 @@
 1: cat
-2: dog
-3: elephant
```

In addition to performing difference comparisons between the current commit pointed to by HEAD and the staged / unstaged changes, another primary use of git diff is to compare differences between the commits in a particular commit history. This is particularly relevant when we are troubleshooting our code: for e.g. if we discover a bug in the most recent commit, we may wish to revert the code base back to an earlier commit where the application was functioning correctly.

To do this we specify the two commits whose contents we wish to compare.

```
git diff commit-ref-A commit-ref-B
```

where *commit-ref-A / B* could either be the hash of the actual commit itself or a branch pointing to a commit. In the comparison output, a will refer to *commit-ref-A*, while b will refer to *commit-ref-B*

When specifying a commit hash, we do not need to specify all 40 characters of its SHA-1 hash. Instead, you just need to specify the first x (where x is typically between 4 - 6) characters that is sufficient to

uniquely identify it from the hashes of all the other Git objects in the repository. However, at the bare minimum you must specify at least 4 characters. Anything less will result in an error.
Other than using the commit hash or a branch name, there are several other ways of referencing a commit, and we will be seeing some of these in an upcoming lab.

Perform a `git log` again and note or copy down the hashes of the last two commits in the commit history. Then compare their contents with:

```
$ git diff last-commit second-last-commit

diff --git a/useless.txt b/useless.txt
new file mode 100644
index 0000000..7ef0c76
--- /dev/null
+++ b/useless.txt
@@ -0,0 +1,5 @@
+just some changes
+
+SD ASDF ASDF AZSDF ASDF
+asdfasdf aqwerqwerqwer
+asdfasdf qwerqwerqwerqwer
\ No newline at end of file
```

Your output may be slightly different, depending on the random changes that you placed in useless.txt earlier. Notice that the --- shows `/dev/null` which indicates the file `useless.txt` did not exist in the first or root commit.

Lets perform a `git diff` on the two most recent commits (where HEAD / MASTER is pointing to the most recent commit):

```
$ git diff 2nd-most-recent-commit master

diff --git a/animals.txt b/animals.txt
index a791d70..8fa46d9 100644
--- a/animals.txt
+++ b/animals.txt
@@ -1,2 +1,3 @@
 1: cat
 2: dog
+3: elephant
diff --git a/cars.txt b/cars.txt
index c18da4e..7dddefe 100644
--- a/cars.txt
+++ b/cars.txt
@@ -1,2 +1,3 @@
 1: honda
 2: mercedes
+3: proton
diff --git a/people.txt b/humans.txt
similarity index 82%
rename from people.txt
rename to humans.txt
index a904830..7f77bbf 100644
--- a/people.txt
```

```
+++ b/humans.txt
@@ -1,2 +1,3 @@
 1: developer
 2: project manager
+3: CEO
```

You can see in the output that the most recent commit adds on a new 3rd line to all the files as well as performs a file rename operation.

5 Configuring the default editor

We will again commit the staged changes that we have, but this time without the `-m` option:

```
$ git commit -a
hint: Waiting for your editor to close the file...
[main 2021-07-08T08:55:37.115Z] update#setState idle
```

This opens up the default editor configured on your system for you to enter the commit message. The file in which the message is entered is `.git/COMMIT_EDITMSG`. You should see a series of comment lines that start like this:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
```

Type this message below those comment lines and save and close the file.

```
Another set of changes
```

You can change the default editor to a different one through a configuration variable, as we have already discussed in a previous lab.

6 Using git log with multiple options

To get a diff comparison between each successive commit in the log history starting from the most recent commit, type:

```
$ git log --patch
```

To get a summary of the diff comparison between each successive commit in the log history starting from the most recent commit, type:

```
$ git log --stat
```

The command `git log` also has many other options available to filter the display of the commit history. As an example:

```
git log commit-ref
```

will list the commit history starting from the specified commit back to the root commit

`git log commit-ref-1..commit-ref-2`

Shows the commit history between the two specified commits, but not inclusive of *commit-ref-1*. For this format to work, *commit-ref-1* must be earlier in the commit history than *commit-ref-2* (i.e. *commit-ref-1* must be an ancestor commit of *commit-ref-2)*

Try the two variations of the `git log` above for yourself by selecting any random commits in the commit history.

To limit the number of commits displayed to x commits, we can type:

`git log -n x`

For e.g.

```
$ git log -n 2
```

will only show the first 2 commits starting from the most recent commit pointed to by HEAD

A particularly useful option to summarize the commit history:

```
$ git log --pretty=oneline
```

The above display is still cluttered with the long hash strings, so we can summarize it further with:

```
$ git log --oneline
```

So far, we have been using the hash of a commit or a branch name to reference a commit.
The ~ relative reference operator is also widely used to traverse backwards in a linear branch to access an ancestor commit relative to specified commit. For e.g. *commit-ref*~3 refers to the commit that is 3 commits behind *commit-ref*, while HEAD~1 refers to the commit immediately behind the one pointed to by HEAD.

For e.g. to list the commit history starting from the 2nd most recent commit back to the root commit:

```
$ git log --oneline master~1
```

To list the 3 most recent commits

```
$ git log --oneline master~3..master
```

You can use the ~ relative reference operator in any place where a commit reference is required for. For e.g. to compare the contents of most recent commit and the 2nd most recent commit, we can use:

```
$ git diff master master~1
```

Let's add a few more lines to animals.txt and stage and commit these changes.

```
2: jaguar
3: elephant
```

Again, we will run the commit command without the `-m` option to start up the default editor to verify that we managed to change the default editor setting correctly from the previous lab:

```
$ git commit -a
hint: Waiting for your editor to close the file...
[main 2021-07-08T08:55:37.115Z] update#setState idle
```

Verify that the editor that starts up is the one that you configured earlier with:

```
git config --global core.editor …….
```

Type this message below the lines of comments and save and close the file.

```
Put in some awesome new animals !
```

Check the two most recent commits in the commit history with:

```
$ git log --oneline -n 2

commit 1764e2d9225022cefda8f6fafddab3fb38f2ee65 (HEAD -> master)
Author: Peter Parker <spiderman@gmail.com>
Date:   Thu Jul 8 16:55:36 2021 +0800

    Put in some awesome new animals !

commit 11606b706570f1496c2234a5115c9d15a616b9eb
Author: Peter Parker <spiderman@gmail.com>
Date:   Thu Jul 8 08:57:26 2021 +0800

    Another set of changes
```

7 Getting help on commands

There are 2 equivalent ways to get help for a particular command,

```
git command --help
git help command
```

This opens up the man page for that command from the documentation stored offline in the Git installation directory (e.g. C:/Program Files/Git/mingw64/share/doc/git-doc). Alternatively, the documentation is available online at: https://git-scm.com/docs

For e.g. type:

```
$ git commit --help
```

This opens up the manual page stored locally at:

file:///C:/Program%20Files/Git/mingw64/share/doc/git-doc/git-commit.html

As you browse through the options, notice that there are sometimes two different ways to specify an option – using a single dash or double dash. Both are equivalent. However not all options provide these two equivalent ways. For e.g.
`git commit -a`
has the same effect as
`git commit --all`

To see the list of argument options available for a command, type:

`git command -h`

For e.g.

```
$ git commit -h
usage: git commit [<options>] [--] <pathspec>...

    -q, --quiet           suppress summary after successful commit
    -v, --verbose         show diff in commit message template

Commit message options
    -F, --file <file>     read message from file
    --author <author>     override author for commit
    --date <date>         override date for commit
    -m, --message <message>
…
```

One important thing to note: we can only run Git commands in a working directory (i.e. a directory that has a Git repository). If we attempt to do this outside of a working directory, we will encounter an error.

For e.g. if we were now to move up to the parent directory and attempt to run `git status`

```
$ cd ..

User@computer MINGW64 /g/labs
$ git status
fatal: not a git repository (or any of the parent directories): .git
```

Once we are outside of a Git working directory, the branch name disappears from the prompt in the shell. This is a warning sign to us. Let's move back again into the working directory again:

```
$ cd firstlab

User@computer MINGW64 /g/labs/firstlab (master)
```