

# Git Lab 2

## Diffs and commit history

1	COMMANDS COVERED .....	1
2	LAB SETUP .....	2
3	PERFORMING DIFF COMPARISONS .....	2
4	USING THE CONFIGURED EDITOR .....	8
5	USING GIT LOG WITH MULTIPLE OPTIONS .....	9
6	GETTING HELP ON COMMANDS .....	11

### 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	<pre>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</pre> <p><a href="#">FreeCodeCamp git log command</a></p> <p><a href="#">Atlassian git log command</a></p>
	git diff

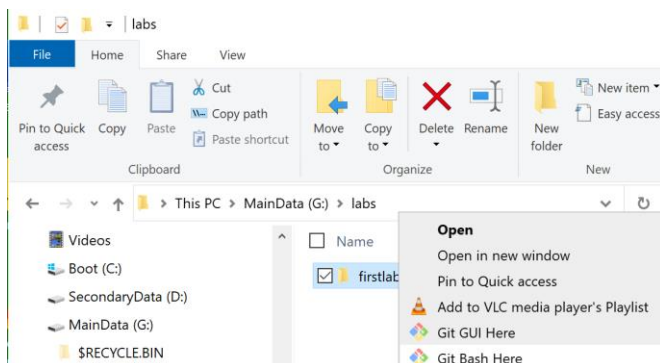
	<pre>git diff --staged git diff <i>commit-hash-A</i>..<i>commit-hash-B</i></pre> <p><a href="#">Atlassian git diff command</a></p> <p><a href="#">git diff complete tutorial</a></p>

## 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 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 3<sup>rd</sup> 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*

To get a diff output that is simpler to interpret, make sure that *commit-ref-A* is earlier than *commit-ref-B* on the commit history time line (i.e. *commit-ref-A* should be an ancestor commit of *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 two earliest commits in the commit history. Then compare their contents with:

```
$ git diff root-commit second-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 3<sup>rd</sup> line to all the files as well as performs a file rename operation.

## 4 Using the configured 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. We configured this earlier at the start of a previous lab session.

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.

## 5 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 timeline 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 2<sup>nd</sup> 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 the 2<sup>nd</sup> most recent commit and the most recent commit, we can use:

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

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 which will start up the default editor to prompt us for a suitable commit message.

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

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

## 6 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>] [--] <paths-spec>...
```

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

Commit message options

<code>-F, --file &lt;file&gt;</code>	read message from file
<code>--author &lt;author&gt;</code>	override author for commit
<code>--date &lt;date&gt;</code>	override date for commit
<code>-m, --message &lt;message&gt;</code>	

...

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)
```