git log --patch or git log –p



**Annotated git log -p Output**

Using the image above, let's do a quick recap of the git log -p output:

* 🔵 - the file that is being displayed
* 🔶 - the hash of the first version of the file and the hash of the second version of the file
  + not usually important, so it's safe to ignore
* ❤️ - the old version and current version of the file
* 🔍 - the lines where the file is added and how many lines there are
  + -15,83 indicates that the old version (represented by the -) started at line 15 and that the file had 83 lines
  + +15,85 indicates that the current version (represented by the +) starts at line 15 and that there are now 85 lines...these 85 lines are shown in the patch below
* ✏️ - the actual changes made in the commit
  + lines that are red and start with a minus (-) were in the original version of the file but have been removed by the commit
  + lines that are green and start with a plus (+) are new lines that have been added in the commit

## Too Much Scrolling

The last few quizzes in the previous section had you scrolling and scrolling through the patch output just to get to the right commit so you could see its info. Wouldn't it be super handy if you could just display a specific commit's details without worrying about all of the others in the repo?

There are actually two ways to do this!

* providing the SHA of the commit you want to see to git log
* use a new command git show

They're both pretty simple, but let's look at the git log way and then we'll look at git show.

You already know how to "log" information with:

* git log
* git log --oneline
* git log --stat
* git log -p

But did you know, you can supply the SHA of a commit as the final argument for all of these commands? For example:

$ git log -p fdf5493

By supplying a SHA, the git log -p command will start at that commit! No need to scroll through everything! Keep in mind that it will also show all of the commits that were made prior to the supplied SHA.

## New Command: git show

The other command that shows a specific commit is git show:

$ git show

Running it like the example above will only display the most recent commit. Typically, a SHA is provided as a final argument:

$ git show fdf5493

### What does git show do?

The git show command will show only one commit. So don't get alarmed when you can't find any other commits - it only shows one. The output of the git show command is exactly the same as the git log -p command. So by default, git show displays:

* the commit
* the author
* the date
* the commit message
* the patch information

However, git show can be combined with most of the other flags we've looked at:

* --stat - to show the how many files were changed and the number of lines that were added/removed
* -p or --patch - this the default, but if --stat is used, the patch won't display, so pass -p to add it again
* -w - to ignore changes to whitespace

## Changes To Be Committed

There's now a new section in the output of git status - the "Changes to be committed" area! This new "Changes to be committed" section displays files that are in the Staging Area! Right now it only displays the index.html file, so this file is the only item on the Staging Index. To continue this train of thought, if we made a commit right now, only the index.html file would be committed.

*TIP: Did you also notice the helpful text that's located just beneath "Changes to be committed"? It says (use "git rm --cached <file>..." to unstage) This is a hint of what you should do if you accidentally ran git add and gave it the wrong file.*

*As a side note, git rm --cached is not like the shell's rm command. git rm --cached will not destroy any of your work; it just removes it from the Staging Index.*

*Also, this used the word "unstage". The act of moving a file from the Working Directory to the Staging Index is called "staging". If a file has been moved, then it has been "staged". Moving a file from the Staging Index*back*to the Working Directory will unstage the file. If you read documentation that says "stage the following files" that means you should use the git add command.*

**Git Add**

The git add command is used to move files from the Working Directory to the Staging Index.

$ git add <file1> <file2> … <fileN>

This command:

* takes a space-separated list of file names
* alternatively, the period . can be used in place of a list of files to tell Git to add the current directory (and all nested files)

## git diff

The git diff command can be used to see changes that have been made but haven't been committed, yet.

$ git diff

This command displays:

* the files that have been modified
* the location of the lines that have been added/removed
* the actual changes that have been made

git log -p uses git diff under the hood.

## Git Ignore

To recap, the .gitignore file is used to tell Git about the files that Git should not track. This file should be placed in the same directory that the .git directory is in.

**Globbing Crash Course**

Let's say that you add 50 images to your project, but want Git to ignore all of them. Does this mean you have to list each and every filename in the .gitignore file? Oh gosh no, that would be crazy! Instead, you can use a concept called [globbing](https://en.wikipedia.org/wiki/Glob_(programming" \t "_blank).

Globbing lets you use special characters to match patterns/characters. In the .gitignore file, you can use the following:

* blank lines can be used for spacing
* # - marks line as a comment
* \* - matches 0 or more characters
* ? - matches 1 character
* [abc] - matches a, b, \_or\_ c
* \*\* - matches nested directories - a/\*\*/z matches
  + a/z
  + a/b/z
  + a/b/c/z

So if all of the 50 images are JPEG images in the "samples" folder, we could add the following line to .gitignore to have Git ignore all 50 images.

### Further Research

* [Ignoring files](https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Ignoring-Files) from the Git Book
* [gitignore](https://git-scm.com/docs/gitignore#_pattern_format) from the Git Docs
* [Ignoring files](https://help.github.com/articles/ignoring-files/) from the GitHub Docs
* [gitignore.io](https://www.gitignore.io/)

## See All Branches At Once

We've made it to the end of all the changes we needed to make! Awesome job!

Now we have multiple sets of changes on three different branches. We can't see other branches in the git log output unless we switch to a branch. Wouldn't it be nice if we could see all branches at once in the git log output.

As you've hopefully learned by now, the git log command is pretty powerful and can show us this information. We'll use the new --graph and --all flags:

$ git log --oneline --decorate --graph –all

The --graph flag adds the bullets and lines to the leftmost part of the output. This shows the actual branching that's happening. The --all flag is what displays all of the branches in the repository.

## ⚠️ Know The Branch ⚠️

It's very important to know which branch you're on when you're about to merge branches together. Remember that making a merge makes a commit.

As of right now, we do not know how to undo changes. We'll go over it in the next lesson, but if you make a merge on the wrong branch, use this command to undo the merge:

git reset --hard HEAD^

(Make sure to include the ^ character! It's a known as a "Relative Commit Reference" and indicates "the parent commit". We'll look at Relative Commit References in the next lesson.)

**The Merge Command**

The git merge command is used to combine Git branches:

$ git merge <name-of-branch-to-merge-in>

When a merge happens, Git will:

* look at the branches that it's going to merge
* look back along the branch's history to find a single commit that *both* branches have in their commit history
* combine the lines of code that were changed on the separate branches together
* makes a commit to record the merge

**What If A Merge Fails?**

The merges we just did were able to merge successfully. Git is able to intelligently combine lots of work on different branches. However, there are times when it can't combine branches together. When a merge is performed and fails, that is called a **merge conflict**. We'll look at merge conflicts, what causes them, and how to resolve them in the next lesson.

**What If A Merge Fails?**

The merges we just did were able to merge successfully. Git is able to intelligently combine lots of work on different branches. However, there are times when it can't combine branches together. When a merge is performed and fails, that is called a **merge conflict**. We'll look at merge conflicts, what causes them, and how to resolve them in the next lesson.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

