Git Status: Inspecting a repository

[**git status**](https://www.atlassian.com/git/tutorials/inspecting-a-repository) [**git tag**](https://www.atlassian.com/git/tutorials/inspecting-a-repository) [**git blame**](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

git status

The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven’t, and which files aren’t being tracked by Git. Status output does *not* show you any information regarding the committed project history. For this, you need to use [git log](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-log).

Related git commands

* [git tag](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-tag)
  + Tags are ref's that point to specific points in Git history. git tag is generally used to capture a point in history that is used for a marked version release (i.e. v1.0.1).
* [git blame](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-blame)
  + The high-level function of git blame is the display of author metadata attached to specific committed lines in a file. This is used to explore the history of specific code and answer questions about what, how, and why the code was added to a repository.
* [git log](https://www.atlassian.com/git/tutorials/git-log)
  + The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes.

Usage

git status

List which files are staged, unstaged, and untracked.

Discussion

The git status command is a relatively straightforward command. It simply shows you what's been going on with git add and git commit. Status messages also include relevant instructions for staging/unstaging files. Sample output showing the three main categories of a git status call is included below:

# On branch main  
# Changes to be committed:  
# (use "git reset HEAD <file>..." to unstage)  
#  
#modified: hello.py  
#  
# 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: main.py  
#  
# Untracked files:  
# (use "git add <file>..." to include in what will be committed)  
#  
#hello.pyc

**Ignoring Files**

Untracked files typically fall into two categories. They're either files that have just been added to the project and haven't been committed yet, or they're compiled binaries like .pyc, .obj, .exe, etc. While it's definitely beneficial to include the former in the git status output, the latter can make it hard to see what’s actually going on in your repository.

For this reason, Git lets you completely ignore files by placing paths in a special file called [.gitignore](https://www.atlassian.com/git/tutorials/gitignore). Any files that you'd like to ignore should be included on a separate line, and the \* symbol can be used as a wildcard. For example, adding the following to a .gitignore file in your project root will prevent compiled Python modules from appearing in git status:

\*.pyc

Example

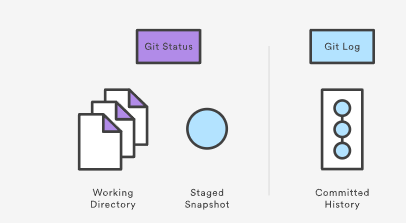
It's good practice to check the state of your repository before committing changes so that you don't accidentally commit something you don't mean to. This example displays the repository status before and after staging and committing a snapshot:

# Edit hello.py  
git status  
# hello.py is listed under "Changes not staged for commit"  
git add hello.py  
git status  
# hello.py is listed under "Changes to be committed"  
git commit  
git status  
# nothing to commit (working directory clean)

The first status output will show the file as unstaged. The git add action will be reflected in the second git status, and the final status output will tell you that there is nothing to commit—the working directory matches the most recent commit. Some Git commands (e.g., [git merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge)) require the working directory to be clean so that you don't accidentally overwrite changes.

git log

The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes. While git status lets you inspect the working directory and the staging area, git log only operates on the committed history.



Log output can be customized in several ways, from simply filtering commits to displaying them in a completely user-defined format. Some of the most common configurations of git log are presented below.

Usage

git log

Display the entire commit history using the default formatting. If the output takes up more than one screen, you can use Space to scroll and q to exit.

git log -n <limit>

Limit the number of commits by . For example, git log -n 3 will display only 3 commits.

Condense each commit to a single line. This is useful for getting a high-level overview of the project history.

git log --oneline

git log --stat

Along with the ordinary git log information, include which files were altered and the relative number of lines that were added or deleted from each of them.

git log -p

Display the patch representing each commit. This shows the full diff of each commit, which is the most detailed view you can have of your project history.

git log --author="<pattern>"

Search for commits by a particular author. The  argument can be a plain string or a regular expression.

git log --grep="<pattern>"

Search for commits with a commit message that matches , which can be a plain string or a regular expression.

git log <since>..<until>

Show only commits that occur between < since > and < until >. Both arguments can be either a commit ID, a branch name, HEAD, or any other kind of [revision reference](http://www.kernel.org/pub/software/scm/git/docs/gitrevisions.html).

git log <file>

Only display commits that include the specified file. This is an easy way to see the history of a particular file.

git log --graph --decorate --oneline

A few useful options to consider. The --graph flag that will draw a text based graph of the commits on the left hand side of the commit messages. --decorate adds the names of branches or tags of the commits that are shown. --oneline shows the commit information on a single line making it easier to browse through commits at-a-glance.

Discussion

The git log command is Git's basic tool for exploring a repository’s history. It’s what you use when you need to find a specific version of a project or figure out what changes will be introduced by merging in a feature branch.

commit 3157ee3718e180a9476bf2e5cab8e3f1e78a73b7  
Author: John Smith

Most of this is pretty straightforward; however, the first line warrants some explanation. The 40-character string after commit is an SHA-1 checksum of the commit’s contents. This serves two purposes. First, it ensures the integrity of the commit—if it was ever corrupted, the commit would generate a different checksum. Second, it serves as a unique ID for the commit.

This ID can be used in commands like git log .. to refer to specific commits. For instance, git log 3157e..5ab91 will display everything between the commits with ID's 3157e and 5ab91. Aside from checksums, branch names (discussed in the [Branch Module](https://www.atlassian.com/git/tutorials/using-branches)) and the HEAD keyword are other common methods for referring to individual commits. HEAD always refers to the current commit, be it a branch or a specific commit.

The ~ character is useful for making relative references to the parent of a commit. For example, 3157e~1 refers to the commit before 3157e, and HEAD~3 is the great-grandparent of the current commit.

The idea behind all of these identification methods is to let you perform actions based on specific commits. The git log command is typically the starting point for these interactions, as it lets you find the commits you want to work with.

Example

The *Usage* section provides many examples of git log, but keep in mind that several options can be combined into a single command:

git log --author="John Smith" -p hello.py

This will display a full diff of all the changes John Smith has made to the file hello.py.

The .. syntax is a very useful tool for comparing branches. The next example displays a brief overview of all the commits that are in some-feature that are not in main.

git log --oneline main..some-feature

# Git tag

[**git status**](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-tag) [**git tag**](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-tag) [**git blame**](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-tag)

## Tagging

This document will discuss the Git concept of tagging and the git tag command. Tags are ref's that point to specific points in Git history. Tagging is generally used to capture a point in history that is used for a marked version release (i.e. v1.0.1). A tag is like a branch that doesn’t change. Unlike branches, tags, after being created, have no further history of commits. For more info on branches visit the git branch page. This document will cover the different kind of tags, how to create tags, listing all tags, deleting tags, sharing tags, and more.

## Creating a tag

To create a new tag execute the following command:

git tag <tagname>

Replace < tagname > with a semantic identifier to the state of the repo at the time the tag is being created. A common pattern is to use version numbers like git tag v1.4. Git supports two different types of tags, annotated and lightweight tags. The previous example created a lightweight tag. Lightweight tags and Annotated tags differ in the amount of accompanying meta data they store. A best practice is to consider Annotated tags as public, and Lightweight tags as private. Annotated tags store extra meta data such as: the tagger name, email, and date. This is important data for a public release. Lightweight tags are essentially 'bookmarks' to a commit, they are just a name and a pointer to a commit, useful for creating quick links to relevant commits.

## Annotated Tags

Annotated tags are stored as full objects in the Git database. To reiterate, They store extra meta data such as: the tagger name, email, and date. Similar to commits and commit messages Annotated tags have a tagging message. Additionally, for security, annotated tags can be signed and verified with GNU Privacy Guard (GPG). Suggested best practices for git tagging is to prefer annotated tags over lightweight so you can have all the associated meta-data.

git tag -a v1.4

Executing this command will create a new annotated tag identified with v1.4. The command will then open up the configured default text editor to prompt for further meta data input.

git tag -a v1.4 -m "my version 1.4"

Executing this command is similar to the previous invocation, however, this version of the command is passed the -m option and a message. This is a convenience method similar to git commit -m that will immediately create a new tag and forgo opening the local text editor in favor of saving the message passed in with the -m option.

## Lightweight Tags

git tag v1.4-lw

Executing this command creates a lightweight tag identified as v1.4-lw. Lightweight tags are created with the absence of the -a, -s, or -m options. Lightweight tags create a new tag checksum and store it in the .git/ directory of the project's repo.

## Listing Tags

To list stored tags in a repo execute the following:

git tag

This will output a list of tags:

v0.10.0  
    v0.10.0-rc1  
    v0.11.0  
    v0.11.0-rc1  
    v0.11.1  
    v0.11.2  
    v0.12.0  
    v0.12.0-rc1  
    v0.12.1  
    v0.12.2  
    v0.13.0  
    v0.13.0-rc1  
    v0.13.0-rc2

To refine the list of tags the -l option can be passed with a wild card expression:

$ git tag -l \*-rc\*  
    v0.10.0-rc1  
    v0.11.0-rc1  
    v0.12.0-rc1  
    v0.13.0-rc1  
    v0.13.0-rc2  
    v0.14.0-rc1  
    v0.9.0-rc1  
    v15.0.0-rc.1  
    v15.0.0-rc.2  
    v15.4.0-rc.3

This previous example uses the -l option and a wildcard expression of -rc which returns a list of all tags marked with a -rc prefix, traditionally used to identify release candidates.

## Tagging Old Commits

The previous tagging examples have demonstrated operations on implicit commits. By default, git tag will create a tag on the commit that HEAD is referencing. Alternatively git tag can be passed as a ref to a specific commit. This will tag the passed commit instead of defaulting to HEAD. To gather a list of older commits execute the git log command.

$ git log --pretty=oneline  
    15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'feature'  
    a6b4c97498bd301d84096da251c98a07c7723e65 add update method for thing  
    0d52aaab4479697da7686c15f77a3d64d9165190 one more thing  
    6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

Executing git log will output a list of commits. In this example we will pick the top most commit Merge branch 'feature' for the new tag. We will need to reference to the commit SHA hash to pass to Git:

git tag -a v1.2 15027957951b64cf874c3557a0f3547bd83b3ff6

Executing the above git tag invocation will create a new annotated commit identified as v1.2 for the commit we selected in the previous git log example.

## ReTagging/Replacing Old Tags

If you try to create a tag with the same identifier as an existing tag, Git will throw an error like:

fatal: tag 'v0.4' already exists

Additionally if you try to tag an older commit with an existing tag identifier Git will throw the same error.

In the event that you must update an existing tag, the -f FORCE option must be used.

git tag -a -f v1.4 15027957951b64cf874c3557a0f3547bd83b3ff6

Executing the above command will map the 15027957951b64cf874c3557a0f3547bd83b3ff6 commit to the v1.4 tag identifier. It will override any existing content for the v1.4 tag.

## Sharing: Pushing Tags to Remote

Sharing tags is similar to pushing branches. By default, git push will not push tags. Tags have to be explicitly passed to git push.

$ git push origin v1.4  
    Counting objects: 14, done.  
    Delta compression using up to 8 threads.  
    Compressing objects: 100% (12/12), done.  
    Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.  
    Total 14 (delta 3), reused 0 (delta 0)  
    To git@bitbucket.com:atlasbro/gittagdocs.git  
     \* [new tag]         v1.4 -> v1.4

To push multiple tags simultaneously pass the --tags option to git push command. When another user clones or pulls a repo they will receive the new tags.

## Checking Out Tags

You can view the state of a repo at a tag by using the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) command.

git checkout v1.4

The above command will checkout the v1.4 tag. This puts the repo in a detached HEAD state. This means any changes made will not update the tag. They will create a new detached commit. This new detached commit will not be part of any branch and will only be reachable directly by the commits SHA hash. Therefore it is a best practice to create a new branch anytime you're making changes in a detached HEAD state.

## Deleting Tags

Deleting tags is a straightforward operation. Passing the -d option and a tag identifier to git tag will delete the identified tag.

$ git tag  
    v1  
    v2  
    v3  
    $ git tag -d v1  
    $ git tag  
    v2  
    v3

In this example git tag is executed to display a list of tags showing v1, v2, v3, Then git tag -d v1 is executed which deletes the v1 tag.

## Summary

To recap, Tagging is an additional mechanism used to create a snap shot of a Git repo. Tagging is traditionally used to create semantic version number identifier tags that correspond to software release cycles. The git tag command is the primary driver of tag: creation, modification and deletion. There are two types of tags; annotated and lightweight. Annotated tags are generally the better practices as they store additional valuable meta data about the tag. Additional Git commands covered in this document were [git push](https://www.atlassian.com/git/tutorials/syncing/git-push), and [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout). Visit their corresponding pages for discussion on their extended use.

# Git blame

[**git status**](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-blame) [**git tag**](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-blame) [**git blame**](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-blame)

The git blame command is a versatile troubleshooting utility that has extensive usage options. The high-level function of git blame is the display of author metadata attached to specific committed lines in a file. This is used to examine specific points of a file's history and get context as to who the last author was that modified the line. This is used to explore the history of specific code and answer questions about what, how, and why the code was added to a repository.

Git blame is often used with a GUI display. Online Git hosting sites like [Bitbucket](https://bitbucket.org/product) offer blame views which are UI wrappers to git blame. These views are referenced in collaborative discussions around pull requests and commits. Additionally, most IDE's that have Git integration also have dynamic blame views.

## How It Works

In order to demonstrate git blame we need a repository with some history. We will use the open source project [git-blame-example](https://bitbucket.org/kevzettler/git-blame-example). This open source project is a simple repository that contains a README.md file which has a few commits from different authors. The first step of our git blame usage example is to git clone the example repository.

git clone https://kevzettler@bitbucket.org/kevzettler/git-blame-example.git && cd git-blame-example

Now that we have a copy of the example code we can start exploring it with git blame. The state of the example repo can be examined using [git log](https://www.atlassian.com/git/tutorials/git-log). The commit history should look like the following:

$ git log  
    commit 548dabed82e4e5f3734c219d5a742b1c259926b2  
    Author: Juni Mukherjee <jmukherjee@atlassian.com>  
    Date:   Thu Mar 1 19:55:15 2018 +0000  
  
        Another commit to help git blame track the who, the what, and the when  
  
    commit eb06faedb1fdd159d62e4438fc8dbe9c9fe0728b  
    Author: Juni Mukherjee <jmukherjee@atlassian.com>  
    Date:   Thu Mar 1 19:53:23 2018 +0000  
  
        Creating the third commit, along with Kev and Albert, so that Kev can get git blame docs.  
  
    commit 990c2b6a84464fee153253dbf02e845a4db372bb  
    Merge: 82496ea 89feb84  
    Author: Albert So <aso@atlassian.com>  
    Date:   Thu Mar 1 05:33:01 2018 +0000  
  
        Merged in albert-so/git-blame-example/albert-so/readmemd-edited-online-with-bitbucket-1519865641474 (pull request #2)  
  
        README.md edited online with Bitbucket  
  
    commit 89feb84d885fe33d1182f2112885c2a64a4206ec  
    Author: Albert So <aso@atlassian.com>  
    Date:   Thu Mar 1 00:54:03 2018 +0000  
  
        README.md edited online with Bitbucket

git blame only operates on individual files. A file-path is required for any useful output. The default execution of git blame will simply output the commands help menu. For this example, we will operate on the README.MD file. It is a common open source software practice to include a README file in the root of a git repository as documentation source for the project.

git blame README.MD

Executing the above command will give us our first sample of blame output. The following output is a subset of the full blame output of the README. Additionally, this output is static is reflective of the state of the repo at the time of this writing.

$ git blame README.md  
    82496ea3 (kevzettler     2018-02-28 13:37:02 -0800  1) # Git Blame example  
    82496ea3 (kevzettler     2018-02-28 13:37:02 -0800  2)  
    89feb84d (Albert So      2018-03-01 00:54:03 +0000  3) This repository is an example of a project with multiple contributors making commits.  
    82496ea3 (kevzettler     2018-02-28 13:37:02 -0800  4)  
    82496ea3 (kevzettler     2018-02-28 13:37:02 -0800  5) The repo use used elsewhere to demonstrate `git blame`  
    82496ea3 (kevzettler     2018-02-28 13:37:02 -0800  6)  
    89feb84d (Albert So      2018-03-01 00:54:03 +0000  7) Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod TEMPOR incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum  
    89feb84d (Albert So      2018-03-01 00:54:03 +0000  8)  
    eb06faed (Juni Mukherjee 2018-03-01 19:53:23 +0000  9) Annotates each line in the given file with information from the revision which last modified the line. Optionally, start annotating from the given revision.  
    eb06faed (Juni Mukherjee 2018-03-01 19:53:23 +0000 10)  
    548dabed (Juni Mukherjee 2018-03-01 19:55:15 +0000 11) Creating a line to support documentation needs for git blame.  
    548dabed (Juni Mukherjee 2018-03-01 19:55:15 +0000 12)  
    548dabed (Juni Mukherjee 2018-03-01 19:55:15 +0000 13) Also, it is important to have a few of these commits to clearly reflect the who, the what and the when. This will help Kev get good screenshots when he runs the git blame on this README.

This is a sample of the first 13 lines of the README.md file. To better understand this output lets break down a line. The following table displays the content of line 3 and the columns of the table indicate the column content.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | Author | Timestamp | Line Number | Line Content |
| 89feb84d | Albert So | 2018-03-01 00:54:03 +0000 | 3 | This repository is an example of a project with multiple contributors making commits. |

If we review the blame output list, we can make some observations. There are three authors listed. In addition to the project's maintainer Kev Zettler, Albert So, and Juni Mukherjee are also listed. Authors are generally the most valuable part of git blame output. The timestamp column is also primarily helpful. What the change was is indicated by line content column.

## Common Options

git blame -L 1,5 README.md

The -L option will restrict the output to the requested line range. Here we have restricted the output to lines 1 through 5.

git blame -e README.md

The -e option shows the authors email address instead of username.

git blame -w README.md

The -w option ignores whitespace changes. If a previous author has modified the spacing of a file by switching from tabs to spaces or adding new lines this, unfortunately, obscures the output of git blame by showing these changes.

git blame -M README.md

The -M option detects moved or copied lines within in the same file. This will report the original author of the lines instead of the last author that moved or copied the lines.

git blame -C README.md

The -C option detects lines that were moved or copied from other files. This will report the original author of the lines instead of the last author that moved or copied the lines.

## Git Blame vs Git Log

While git blame displays the last author that modified a line, often times you will want to know when a line was originally added. This can be cumbersome to achieve using git blame. It requires a combination of the -w, -C, and -M options. It can be far more convenient to use the [git log](https://www.atlassian.com/git/tutorials/git-log) command.

To list all original commits in-which a specific code piece was added or modified execute git log with the -S option. Append the -S option with the code you are looking for. Let's take one of the lines from the README output above to use as an example. Let us take the text "CSS3D and WebGL renderers" from Line 12 of the README output.

$ git log -S"CSS3D and WebGL renderers." --pretty=format:'%h %an %ad %s'  
    e339d3c85 Mario Schuettel Tue Oct 13 16:51:06 2015 +0200 reverted README.md to original content  
    509c2cc35 Daniel Tue Sep 8 13:56:14 2015 +0200 Updated README  
    cb20237cc Mr.doob Mon Dec 31 00:22:36 2012 +0100 Removed DOMRenderer. Now with the CSS3DRenderer it has become irrelevant.

This output shows us that content from the README was added or modified 3 times by 3 different authors. It was originally added in commit cb20237cc by Mr.doob. In this example, git log has also been prepended with the --pretty-format option. This option converts the default output format of git log into one that matches the format of git log. For more information on usage and configuration options visit the [git log](https://www.atlassian.com/git/tutorials/git-log) page.

## Summary

The git blame command is used to examine the contents of a file line by line and see when each line was last modified and who the author of the modifications was. The output format of git blame can be altered with various command line options. Online Git hosting solutions like Bitbucket offer blame views, which offer a superior user experience to command line git blame usage. git blame and git log can be used in combination to help discover the history of a file's contents. The git log command has some similar blame functionality, to learn more visit the [git log](https://www.atlassian.com/git/tutorials/git-log) overview page.