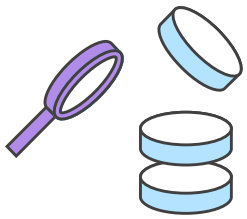


Inspecting a repository

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`.

Usage

```
git status
```

Tutorials

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 master
# 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 c
# (use "git checkout -- <file>..." to discard chang
#
#modified: main.py
#
# Untracked files:
# (use "git add <file>..." to include in what will
#
#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`. Any files that you'd like to ignore should be included on a separate line, and the * symbol can be used as a

```
*.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:

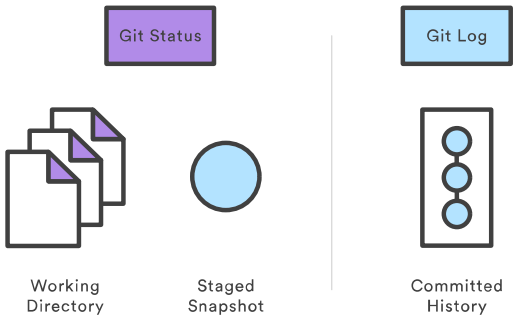
```
# Edit hello.py
git status
# hello.py is listed under "Changes not staged fo
git add hello.py
git status
# hello.py is listed under "Changes to be committ
git commit
git status
# nothing to commit (working directory clean)
```

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`) 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

Tutorials



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 `<limit>`. For example, `git log -n 3` will display only 3 commits.

```
git log --oneline
```

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

Tutorials

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 `<pattern>` argument can be a plain string or a regular expression.

```
git log --grep="<pattern>"
```

Search for commits with a commit message that matches `<pattern>`, 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.

```
git log <file>
```

Only display commits that include the specified file. This

Tutorials

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 <since>..<until>` to refer to specific commits. For instance, `git log 3157e..5ab91` will display everything between the commits with ID's `3157e` and

Tutorials

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

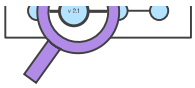
The *Usage* section provides many examples of `git log`, but keep in mind that several options can be combined into a single command:

```
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 `master`.

```
git log --oneline master..some-feature
```



Next up:

Viewing old commits

START NEXT TUTORIAL

Powered By

Tutorials

Enter Your Email For Git News

Getting Started

Setting up a repository

Saving changes

Inspecting a repository

git status

git log

Viewing old commits

Undoing Changes

Rewriting history

Collaborating

Migrating to Git

Advanced Tips