

# Checking the Status of Your Files

The main tool you use to determine which files are in which state is the `git status` command. If you run this command directly after a clone, you should see something like this:

```
git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

This means you have a clean working directory; in other words, none of your tracked files are modified. Git also doesn't see any untracked files, or they would be listed here. Finally, the command tells you which branch you're on and informs you that it has not diverged from the same branch on the server. For now, that branch is always "master", which is the default; you won't worry about it here. [Git Branching](#) will go over branches and references in detail.

Let's say you add a new file to your project, a simple `README` file. If the file didn't exist before, and you run `git status`, you see your untracked file like so:

```
echo 'My Project' > README
git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

  README

nothing added to commit but untracked files present (use "git add" to track)
```

You can see that your new `README` file is untracked, because it's under the "Untracked files" heading in your status output. Untracked basically means that Git sees a file you didn't have in the previous snapshot (commit); Git won't start including it in your commit snapshots until you explicitly tell it to do so. It does this so you don't accidentally begin including generated binary files or other files that you did not mean to include. You do want to start including `README`, so let's start tracking the file.

## Tracking New Files

In order to begin tracking a new file, you use the command `git add`. To begin tracking the `README` file, you can run this:

```
git add README
```

If you run your status command again, you can see that your `README` file is now tracked and staged to be committed:

```
git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   new file:   README
```

You can tell that it's staged because it's under the "Changes to be committed" heading. If you commit at this point, the version of the file at the time you ran `git add` is what will be in the subsequent historical snapshot. You may recall that when you ran `git init` earlier, you then ran `git add <files>` — that was to begin tracking files in your directory. The `git add` command takes a path name for either a file or a directory; if it's a directory, the command adds all the files in that directory recursively.

## Staging Modified Files

Let's change a file that was already tracked. If you change a previously tracked file called `CONTRIBUTING.md` and then run your `git status` command again, you get something that looks like this:

```
git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   new file:   README

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: CONTRIBUTING.md
```

The `CONTRIBUTING.md` file appears under a section named “Changes not staged for commit” — which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the `git add` command. `git add` is a multipurpose command — you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as “add precisely this content to the next commit” rather than “add this file to the project”. Let’s run `git add` now to stage the `CONTRIBUTING.md` file, and then run `git status` again:

```
git add CONTRIBUTING.md
git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   new file:   README
   modified:  CONTRIBUTING.md
```

Both files are staged and will go into your next commit. At this point, suppose you remember one little change that you want to make in `CONTRIBUTING.md` before you commit it. You open it again and make that change, and you’re ready to commit. However, let’s run `git status` one more time:

```
vim CONTRIBUTING.md
git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   new file:   README
   modified:  CONTRIBUTING.md

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:  CONTRIBUTING.md
```

What the heck? Now `CONTRIBUTING.md` is listed as both staged *and* unstaged. How is that possible? It turns out that Git stages a file exactly as it is when you run the `git add` command. If you commit now, the version of `CONTRIBUTING.md` as it was when you last ran the `git add` command is how it will go into the commit, not the version of the file as it looks in your working directory when you run `git commit`. If you modify a file after you run `git add`, you have to run `git add` again to stage the latest version of the file:

```
git add CONTRIBUTING.md
git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

## Short Status

While the `git status` output is pretty comprehensive, it's also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run `git status -s` or `git status --short` you get a far more simplified output from the command:

```
git status -s
M README
M Rakefile
 lib/git.rb
 lib/simplegit.rb
? LICENSE.txt
```

New files that aren't tracked have a `??` next to them, new files that have been added to the staging area have an `A`, modified files have an `M` and so on. There are two columns to the output — the left-hand column indicates the status of the staging area and the right-hand column indicates the status of the working tree. So for example in that output, the `README` file is modified in the working directory but not yet staged, while the `lib/simplegit.rb` file is modified and staged. The `Rakefile` was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

Revision #1

Created 16 April 2019 19:55:01 by ClassCloud

Updated 16 April 2019 20:33:57 by ClassCloud