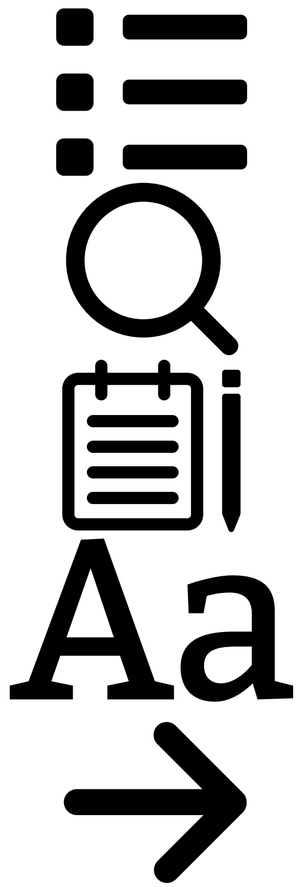


4
The Staging Area Written by Bhagat Singh & Chris Belanger

In previous chapters, you've gained some knowledge of the staging area of Git: You've learned how to stage modifications to your files, stage the addition of new files to the repository, view diffs between your working tree and the staging area, and you even got a little taste of how `git log` works.
But there's more to the staging area than just those few operations. At this point, you may be wondering why the staging area is necessary. "Why can't you just push all of your current updates to the repository directly?", you may ask. It's a good question, but there are issues with that linear approach; Git was actually designed to solve some of the common issues with direct-commit history that exist under other version control systems.
In this chapter, you'll learn a bit more about how the staging area of Git works, why it's necessary, how to undo changes you've made to the staging area, how to move and delete files in your repository, and more.

Why staging exists

Development is a messy process. What, in theory, should be a linear, cumulative construction of functionality in code, is more often than not a series of intertwining, non-linear threads of dead-end code, partly finished features, stubbed-out tests, collections of `// TODO:` comments in the code, and other things that are inherent to a human-driven and largely hand-crafted process.
It's noble to think that that you'll work on just one feature or bug at a time; that your working tree will only ever be populated with clean, fully documented code; that you'll never have unnecessary files cluttering up your working tree; that the configuration of your development environment will always be in perfect sync with the rest of your team; and that you won't follow any rabbit trails (or create a few of your own) while you're investigating a bug.
Git was built to compensate for this messy, non-linear approach to development. It's possible to work on *lots* of things at once, and selectively choose what you want to stage and commit to the repository. The general philosophy is that a commit should be a logical collection of changes *that make sense as a unit* — not just "the latest collection of things I updated that may or may not be related."

A simple staging example

In the example below, I'm working on a website, and I want my design guru to review my CSS changes. I've changed the following files in the course of my work:

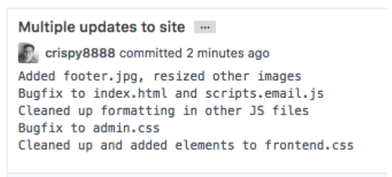
```
index.html

images/favicon.ico
images/header.jpg
images/footer.jpg
images/profile.jpg
```

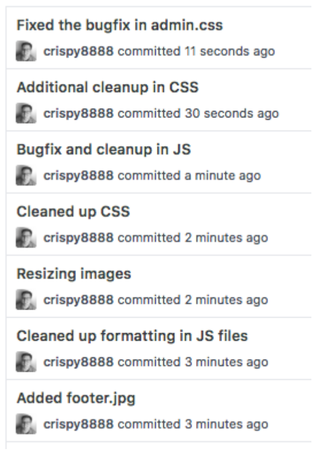
```
styles/admin.css
styles/frontend.css

scripts/main.js
scripts/admin.js
scripts/email.js
```

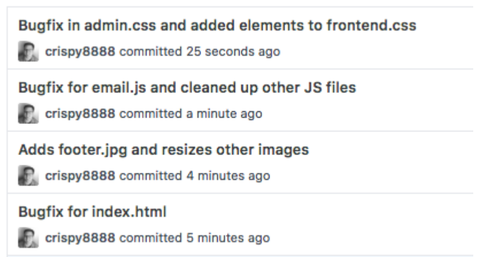
I've updated a bunch of files, here, not just the CSS. And if I had to commit *everything* I had changed in my working directory, all at once, I'd have everything jammed into one commit:



And if I committed each little change as I made it, my commit history might look like the following:



Then, when my design guru wants to take a look at the CSS changes, she'll have to wade through my commit messages and potentially look through my diffs, or even ping me on Slack to figure out what files she's supposed to review.
But, instead, if I were to stage and commit the HTML change first, followed by the image changes, followed by the JavaScript changes, and then the CSS changes after that, the commit history, and even the mental picture of what I did, becomes a *lot* more clear:



In later chapters of the book, you'll come to understand the power of being able to consciously choose various changes to stage for commit, and even choose just a portion of a file to stage for commit. But, for now, you'll explore a few more common scenarios, involving moving files, deleting files, and even undoing your changes that you weren't *quite* ready to commit.

Undoing staged changes

It's quite common that you'll change your mind about a particular set of staged changes, or you might even use something like `git add .` and then realize that there was something in there you didn't quite want to stage.
You've got a file already for book ideas, but you also want to capture some ideas for non-technical management books. Not *everyone* wants to learn how to program, it seems.
Head back to your terminal program, and create a new file in the **books** directory, named **management_book_ideas.md**:

```
touch books/management_book_ideas.md
```

But, wait — the video production team pings you and urgently requests that you update the video content ideas file, since they've just found someone to create the "Getting started with Symbian" course, and, oh, could you also add, "Advanced MOS 6510 Programming" to the list?
OK, not a huge issue. Open up **videos/content_ideas.md**, mark the "Getting started with Symbian" entry as complete by putting an "x" between the brackets, and add a line to the end for the "Advanced MOS 6510 Programming" entry. When you're done, your file should look like this:

```
# Content Ideas

Suggestions for new content to appear as videos:

[x] Beginning Pascal
[ ] Mastering Pascal
[x] Getting started with Symbian
[ ] Coding for the Psion V
[ ] Flash for developers
```

```
[ ] Advanced MOS 6510 Programming
```

Now, execute the following command to add those recent changes to your staging area:
`git add .`

Execute the following command to see what Git thinks about the current state of things:
`git status`

You should see the following:
```
On branch main
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  new file:   books/management_book_ideas.md
  modified:   videos/content_ideas.md
```

Oh, crud. You accidentally added that empty **books/management_book_ideas.md**. You likely didn't want to commit that file just yet, did you? Well, now you're in a pickle. Now that something is in the staging area, how do you get rid of it?
Fortunately, since Git understands everything that's changed so far, it can easily revert your changes for you. The easiest way to do this is through `git reset`.

git reset

Execute the following command to remove the change to **books/management_book_ideas.md** from the staging area:
`git reset HEAD books/management_book_ideas.md`

`git reset` restores your environment to a particular state. But wait — what's this `HEAD` business?
`HEAD` is simply a label that references the most recent commit. You may have already noticed the term `HEAD` in your console output while working through earlier portions of the book.
In case you missed it, execute the following command to look at the log:
`git log`

If you look at the top lines of the output in your console, you'll see something similar to the following:
```
commit 6c88142dc775c4289b764cb9cf2e644274072102 (HEAD -> main)
Author: Chris Belanger <chris@razeware.com>
Date:   Sat Jan 19 07:16:11 2019 -0400

    Adding some tutorial ideas
```

That `(HEAD -> main)` note tells you that the latest commit on your local system is as you expect — the commit where you added those tutorial ideas — and that this commit was done on the `main` branch. You'll get into branches a little later in this section, but, for now, simply understand that `HEAD` keeps track of your latest commit.
So, `git reset HEAD books/management_book_ideas.md`, in this context means "use `HEAD` as a reference point, restore the staging area to that point, but only restore any changes related to the **books/management_book_ideas.md** file."
To see that this is actually the case, exit out of `git log` with `Q` if necessary, and execute `git status` once again:
```
~/GitApprentice/ideas $ git status
On branch main
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  modified:   videos/content_ideas.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  books/management_book_ideas.md
```

That looks better: Git is no longer tracking **books/management_book_ideas.md**, but it's still tracking your changes to **videos/content_ideas.md**. Phew — you're back to where you wanted to be.
Better commit that last change before you get into more trouble. Execute the following command to add another commit:
`git commit -m "Updates book ideas for Symbian and MOS 6510"`

Now, you've been thinking a bit, and you don't think you should keep those ideas about the video platform itself in the **videos** folder. They more appropriately belong in a new folder: **website**.

Moving files in Git

Create the folder for the website ideas with the following command:
`mkdir website`

Now, you need to move that file from the **videos** directory to the **website** directory. Even with your short experience with Git, you probably suspect that it's not quite as simple as just moving the file from one directory to the other. That's correct, but it's instructive to see *why* this is.
So, you'll move it the brute force way first, and see how Git interprets your actions. Execute the following command to use the standard `mv` command line tool to move the file from one directory to the other:
`mv videos/platform_ideas.md website/`

Now, execute `git status` to see what Git thinks about what you've done:
```
~/GitApprentice/ideas $ git status
On branch main
Your branch is ahead of 'origin/main' by 4 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
```

```
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
  deleted:    videos/platform_ideas.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  books/management_book_ideas.md
  website/

no changes added to commit (use "git add" and/or "git commit -a")
```

Well, that's a bit of a mess. Git thinks you've deleted a file that is being tracked, and it also thinks that you've added this **website** bit of nonsense. Git doesn't seem so smart after all. Why doesn't it just *see* that you've moved the file?
The answer is in the way that Git thinks about files: as full paths, not individual directories. Take a look at how Git saw this part of the working tree before the move:

```
videos/platform_ideas.md (tracked)
videos/content_ideas.md (tracked)
```

And, after the move, here's what it sees:

```
videos/platform_ideas.md (deleted)
videos/content_ideas.md (tracked)
website/platform_ideas.md (untracked)
```

Remember, Git knows nothing about directories: It only knows about full paths. Comparing the two snippets of your working tree above shows you exactly why `git status` reports what it does.
Seems like the brute force approach of `mv` isn't what you want. Git has a built-in `mv` command to move things "properly" for you.
Move the file back with the following command:

```
mv website/platform_ideas.md videos/
```

Now, execute the following:

```
git mv videos/platform_ideas.md website/
```

And execute `git status` to see what's up:

```
~/GitApprentice/ideas $ git status
On branch main
Your branch is ahead of 'origin/main' by 4 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  renamed:    videos/platform_ideas.md -> website/platform_ideas.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  books/management_book_ideas.md
```

That looks better. Git sees the file as "renamed," which makes sense, since Git thinks about files in terms of their full path. And Git has also staged that change for you. Nice!
Commit those changes now:

```
git commit -m "Moves platform ideas to website directory"
```

Your ideas project is now looking pretty ship-shape. But, to be honest, those live streaming ideas are pretty bad. Perhaps you should just get rid of them now before too many people see them.

Deleting files in Git

The impulse to just delete/move/rename files as you'd normally do on your filesystem is usually what puts Git into a tizzy, and it causes people to say they don't "get" Git. But if you take the time to instruct Git on what to do, it usually takes care of things quite nicely for you.
So — that live streaming ideas file has to go. The brute-force approach, as you may guess, isn't the best way to solve things, but let's see if it causes Git any grief.
Execute the following command to delete the live streaming ideas file with the `rm` command:

```
rm articles/live_streaming_ideas.md
```

And then execute `git status` to see what Git's reaction is:

```
~/GitApprentice/ideas $ git status
On branch main
Your branch is ahead of 'origin/main' by 5 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
  deleted:    articles/live_streaming_ideas.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  books/management_book_ideas.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Oh, that's not so bad. Git recognizes that you've deleted the file and is prompting you to stage it.
Do that now with the following command:

```
git add articles/live_streaming_ideas.md
```

Then, see what's up with `git status`:

```
~/GitApprentice/ideas $ git status
On branch main
Your branch is ahead of 'origin/main' by 5 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  deleted:    articles/live_streaming_ideas.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  books/management_book_ideas.md
```

Well, that was a bit of a roundabout way to do things. But just like `git mv`, you can use the `git rm` command to do this in one fell swoop.

Restoring deleted files

First, you need to get back to where you were. Unstage the change to the live streaming ideas file with your best new friend, `git reset`:

```
git reset HEAD articles/live_streaming_ideas.md
```

Git tells you explicitly that it has unstaged the files post the `reset` that you just did.

```
Unstaged changes after reset:
D       articles/live_streaming_ideas.md
```

That removes that change from the staging area — but it doesn't *restore* the file itself in your working tree. To do that, you'll need to tell Git to retrieve the latest committed version of that file from the repository.
Execute the following to restore your file to its original infamy:

```
git checkout HEAD articles/live_streaming_ideas.md
```

You're back to where you started. You can verify that the file was restored by what Git tells you after executing the above command:

```
Updated 1 path from b80985f
```

Now, get rid of that file with the following command:

```
git rm articles/live_streaming_ideas.md
```

And, finally, commit that change with an appropriate message:

```
git commit -m "Removes terrible live streaming ideas"
```

Looks like you'll have to leave the live streaming to the experts: fourteen-year-olds on YouTube with too much time on their hands and too little common sense.
That empty file for management book ideas is still hanging around. Since you don't have any good ideas for that file yet, you may as well commit it and hope that someone down the road can populate it with good ways to be an effective manager.
Add that empty file with the following command:

```
git add books/management_book_ideas.md
```

And commit it with a nice comment:

```
git commit -m "Adds all the good ideas about management"
```

It's not all bad: Abandoning your attempts to building a career in live streaming *and* management gives you more time to take on this next challenge!

Challenge

Challenge: Move, delete and restore a file

This challenge takes you through the paces of what you just learned. You'll need to do the following:

. Move the newly added **books/management_book_ideas.md** to the **website** directory with the `git mv` command.

. You've changed your mind and don't want **management_book_ideas.md** anymore, so remove that file completely with the `git rm` command. Git will give you an error when you do this, but look at the suggested actions in the error closely to see how to solve this problem this with the `-f` option, and try again.

. But now you're having second thoughts: Maybe you *do* have some good ideas about management. Restore that file to its original location.

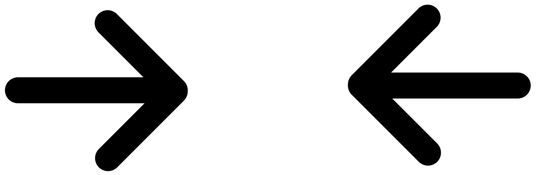
Remember to use the `git status` command to get your bearings when you need to. Liberal use of `git status` will definitely help you understand what Git is doing at each stage of this challenge.
If you get stuck, or want to check your solution, you can always find the answer to this challenge under the **challenge** folder for this chapter.

Key points

The **staging area** lets you construct your next commit in a logical, structure fashion.
`git reset HEAD <filename>` lets you restore your staging environment to the last commit state.
Moving files around and deleting them from the filesystem, without notifying Git, will cause you grief.
`git mv` moves files around and stages the change, all in one action.
`git rm` removes files from your repository and stages the change, again, in one action.
Restore deleted and staged files with `git reset HEAD <filename>` followed by `git checkout HEAD <filename>`

Where to go from here?

That was quite a ride! You've gotten deeper into understanding how Git sees the world; building up a parallel mental model will help you out immensely as you use Git more in your daily workflow.
Sometimes, you may have files that you explicitly *don't* want to add to your repository, but that you want to keep around in your working tree. You can tell Git to ignore things in your working tree, and even tell Git to ignore particular files across *all* of your projects through the magic of the simple file known as **.gitignore** — which you'll learn all about in the next chapter!



5. Ignoring Files in Git 3. Committing Your Changes

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.

© 2022 Razeware LLC