The Only Git Tutorial You Will Ever Need

Repository Links

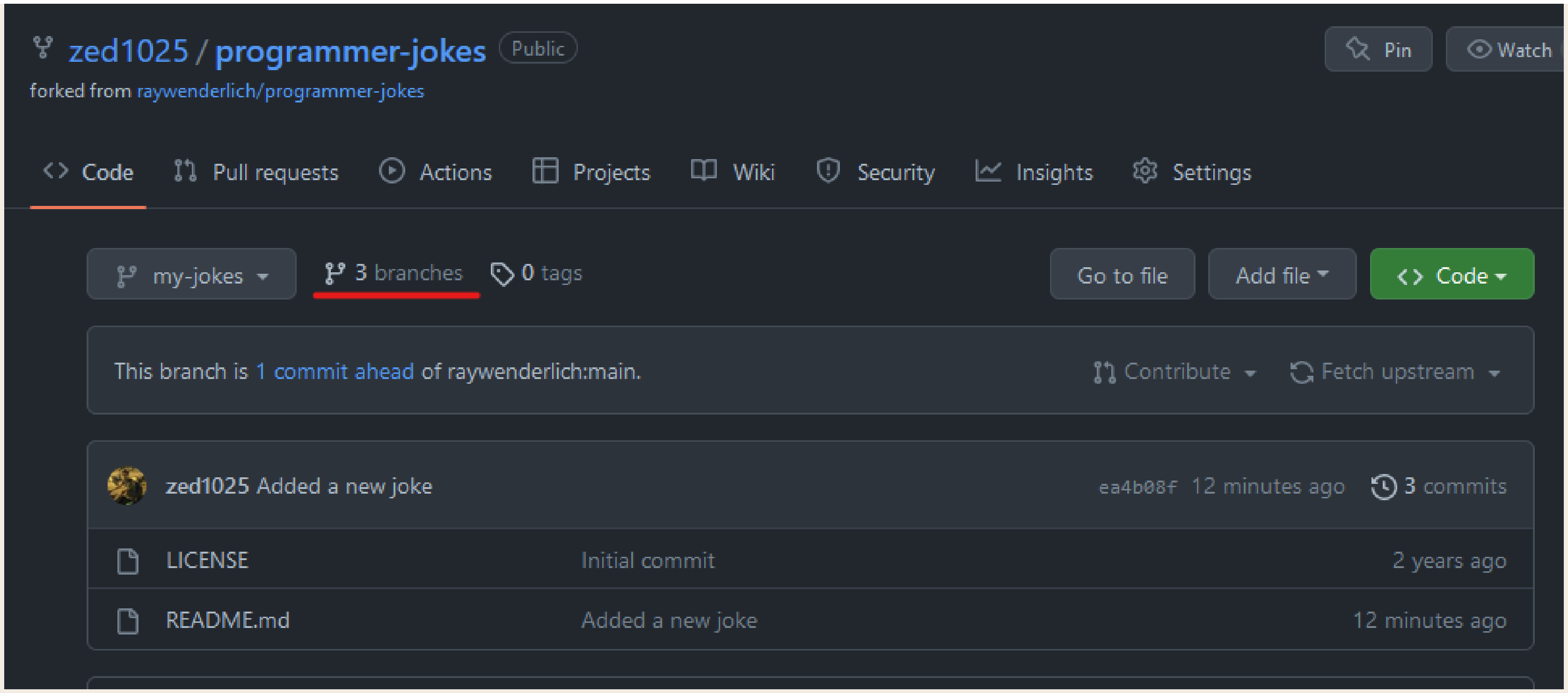
- https://github.com/zed1025/programmer-jokes
- https://github.com/zed1025/ideas

A Simple Git-GitHub Workflow

A Simple Git-GitHub Workflow

1. Go to this URL: https://github.com/raywenderlich/programmer-jokes
2. Click the Fork button
3. Copy the URL of the Forked repository

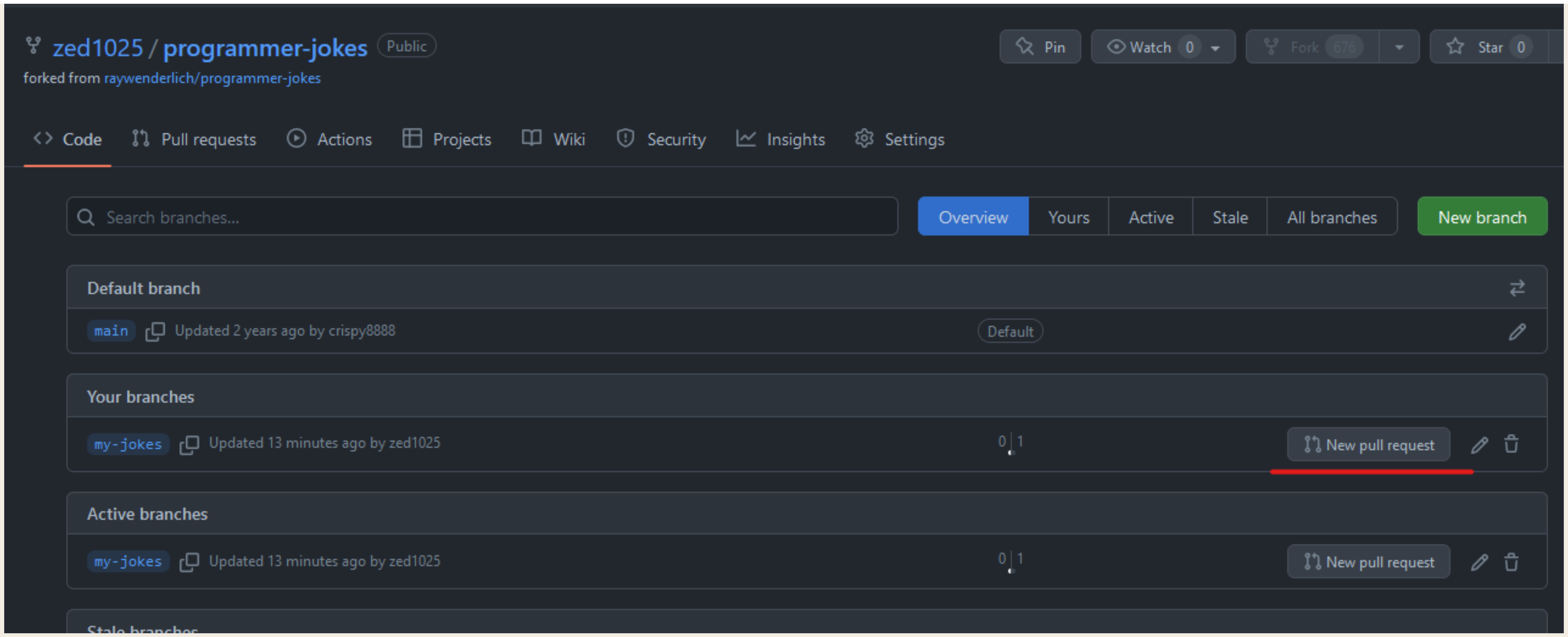
4. Clone the repository in your local system, `git clone https://github.com/zed1025/programmer-jokes.git`
5. Create a new branch `git branch my-joke`
6. See the list of all branched `git branch`
7. Switch to the newly created branch `git checkout my-joke`
8. Add the following lines to README.md. *Why couldn't the confirmed bachelor use Git? Because he was afraid to commit!*
9. Flag the changes made to the file using `git add README.md`. This adds the file to the staging area. (Working Directory -> Staging Area -> Git Repository)
10. Next we want to commit the changes. Committing is the act of saying, "Yes, I have these changes ready, and I want to formally record those changes in my local copy of the repository."
11. Commit the changes with the following command `git commit -m "Added a new joke"`
12. Git has formally recorded the changes. Now it's time to push these changes to the remote repository. `git push --set-upstream origin my-joke`
    a. **push** tells Git to put your local changes on the server
    b. **--set-upstream** tells Git to form a tracking link for this branch between your local repository and the remote repository
    c. **origin** is a convention that references the remote repository
    d. **my-joke** is the branch you want to push
13. There's one thing left to do - Signal to anyone else using this repository that you have something you'd like to integrate, or pull, into the remote repository. You do that with a mechanism called a **pull request**.
14. Open your forked repository on GitHub
15. You'll see that you now have 2 branches(maybe more, but only the branch you created *my-jokes* is important to you)


zed1025 / programmer-jokes Public
forked from raywenderlich/programmer-jokes

This branch is 1 commit ahead of raywenderlich:main.

zed1025 Added a new joke	ea4b08f 12 minutes ago	3 commits
LICENSE	Initial commit	2 years ago
README.md	Added a new joke	12 minutes ago

16. Click on the text saying "2 branches"
17. Here you will see the list of all the branches along with *my-jokes*. Click on the **New pull request** button next to *my-jokes*



18. Click that button and you'll be taken to another page, where you can enter some details about your change. Enter Adds a real knee-slapper to the large text box to give some extra detail about the changes contained in this pull request, then click the Create pull request button
19. END

Cloning a repo

What is cloning?

Cloning is exactly what it sounds like: creating a copy, or clone, of a repository. A Git repository tracks the history of all changes inside the repository through a hidden *.git* directory.

So since a Git repository is just a special directory, you could, in theory, effect a pretty cheap and dirty clone operation by zipping up all the files in a repository on your friend's or colleague's workstation and then emailing it to yourself. When you extract the contents of that zipped-up file, you'd have an exact copy of the repository on your computer.

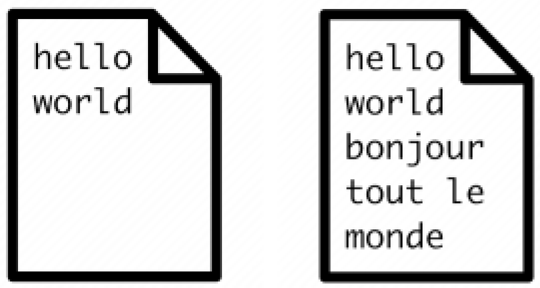
However, emailing things around can (and does) get messy. Instead, many organizations make use of online repository hosts, such as GitHub, GitLab, BitBucket or others. Some organizations even choose to self-host repositories.

Committing your changes

What is a commit?

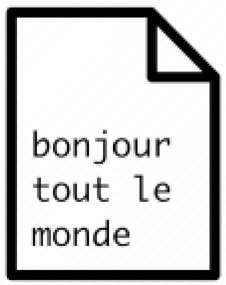
It is essentially a snapshot of the particular state of the set of files in the repository at a point in time.

The state of the repository before you began those updates — your starting point, in effect — is the **parent** commit. After you commit your changes — which is the **diff** — that next commit would be the **child** commit. The diagram below explains this a little more:



Example of two commits, the parent on the left, and the child on the right.

In this example, I've added new text to a file between commits. The parent commit is the left-hand file, and the child commit is the right-hand file. The diff between them are the changes I made to a single file:



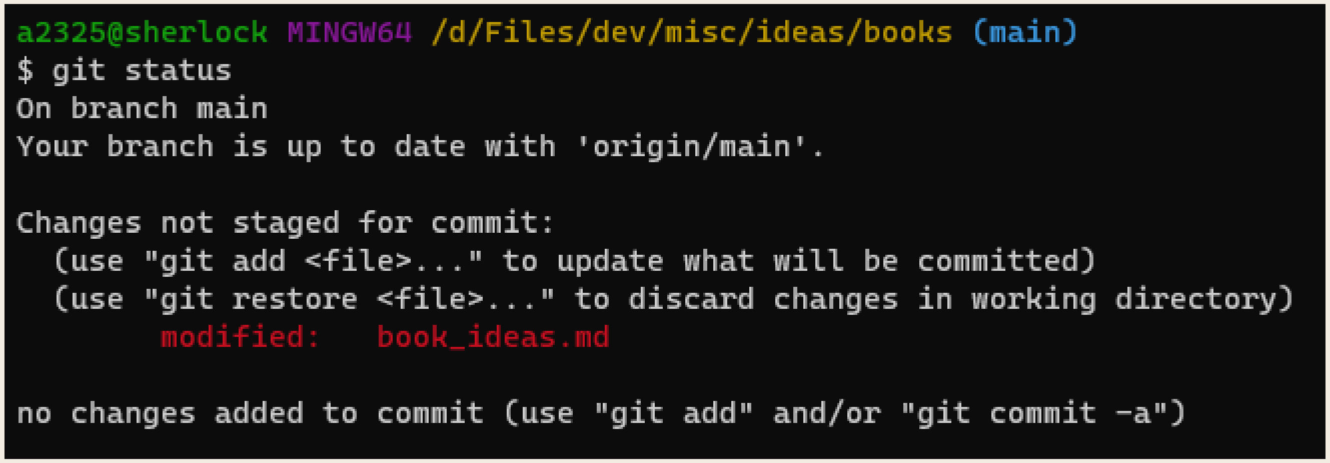
The diff is the difference between the above two commits.

And a diff doesn't just have to be additions to files; creating new content, modifying content and deleting content are other common changes that you'll make to the files in your repository.

Working trees and staging areas

The *working copy* or *working tree* or *working directory* (language is great, there's always more than one name for something) is the collection of project files on your disk that you work with and modify directly.
Git thinks about the files in your working tree as being in three distinct states

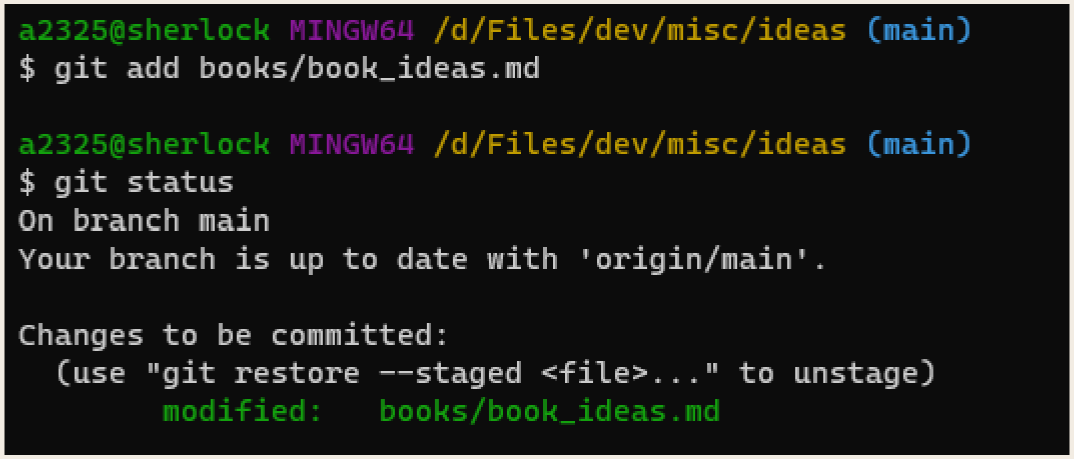
- **unmodified**: Unmodified simply means that you haven't changed this file since your last commit.
- **modified**: Modified is simply the opposite of that: Git sees that you've modified this file in some fashion since your last commit.
- **staged**: Essentially, as you work on bits and pieces of your project, you can mark a change, or set of changes, as "staged," which is how you tell Git, "Hey, I want these changes to go into my next commit… but I might have some more changes for you, so just hold on to these changes for a bit." You can add and remove changes from this staging area as you go about your work, and only commit that set of carefully curated changes to the repository when you're good and ready.
  Notice above that I said, "Add and remove changes from the staging area," not "Add and remove files from the staging area." There's a distinct difference
- I added a new line "- [] Care and feeding of developers" to the file book_ideas.md
- Run the `git status` command



```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas/books (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

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

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

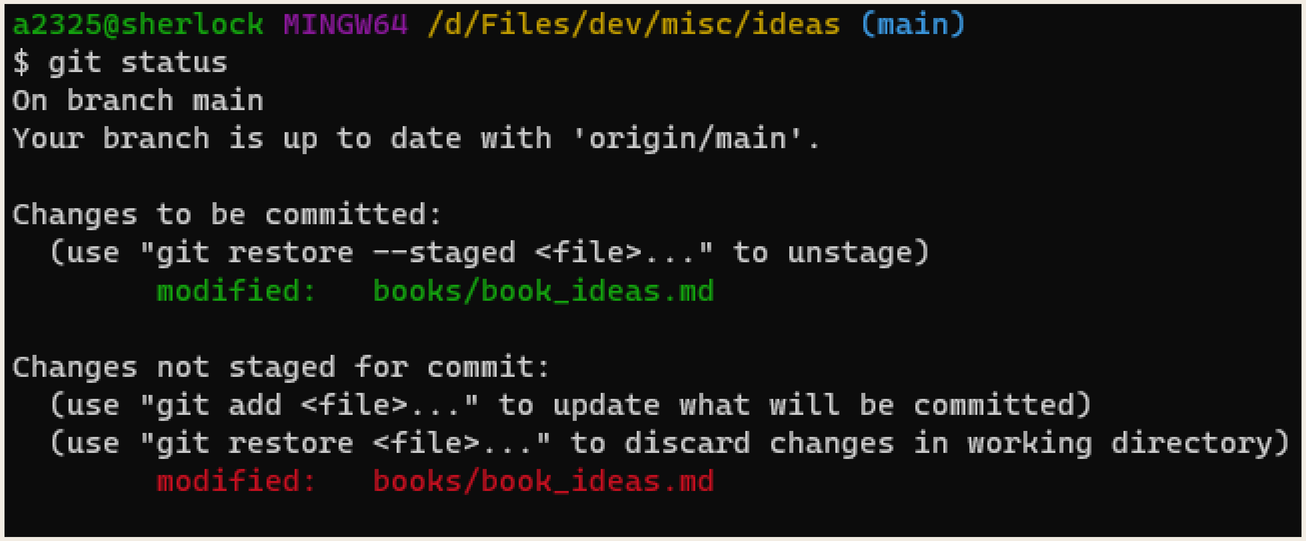
- Stage the changes using `git add books/book_ideas.md`

- Again run `git status`



```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git add books/book_ideas.md

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

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

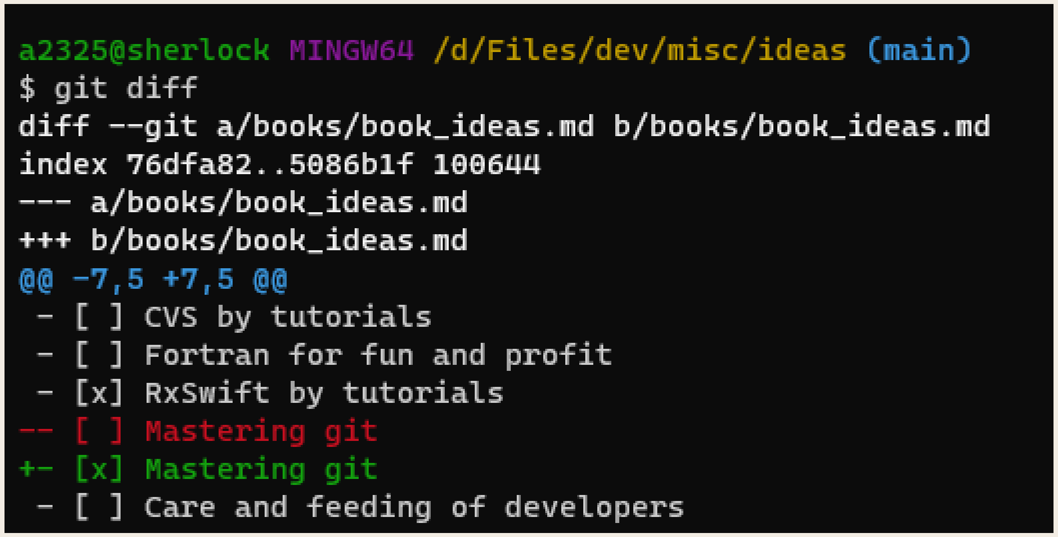
- Git recognizes that you've now placed this change in the staging area. But you have another modification to make to this file that you forgot about. Open books/book_ideas.md in your text editor and make the necessary changes. Then run `git status`



```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

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

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   books/book_ideas.md
```

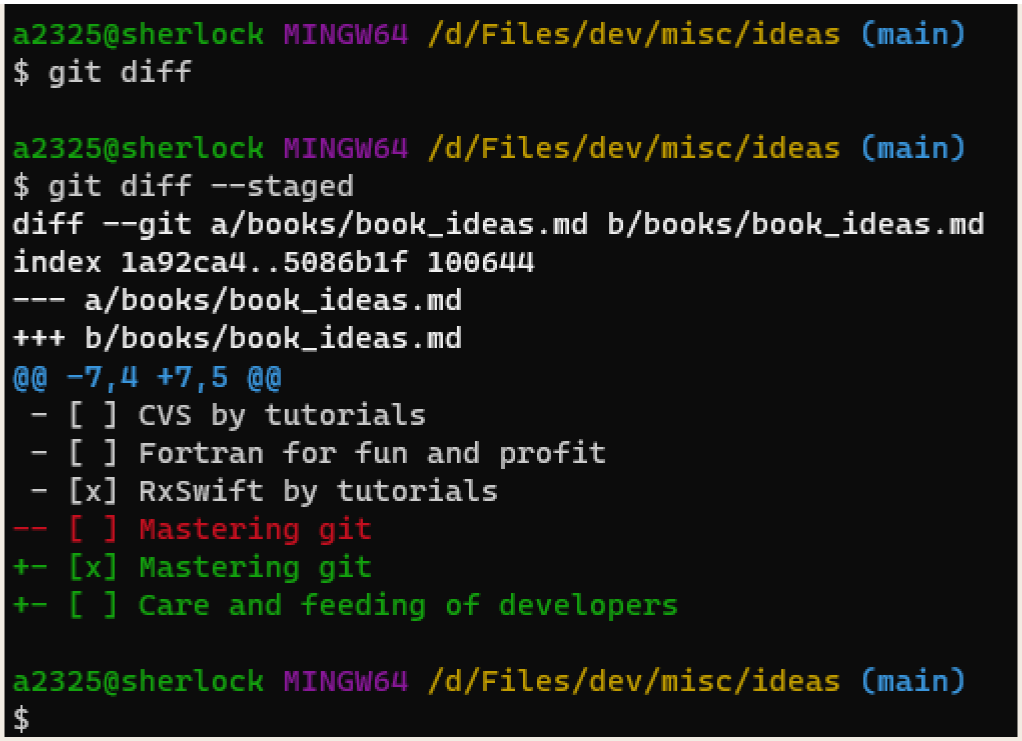
- What gives? Git now tells you that books/book*ideas.md is both staged and not staged? How can that be? _Remember that you're staging changes here, not files*. Git understands this, and tells you that you have one change already staged for commit, and that you have one change that's not yet been staged.

- To see this in detail, you can tell Git to show you what it sees as changed. Run `git diff`



```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git diff
diff --git a/books/book_ideas.md b/books/book_ideas.md
index 76dfa82..5086b1f 100644
--- a/books/book_ideas.md
+++ b/books/book_ideas.md
@@ -7,5 +7,5 @@
 - [ ] CVS by tutorials
 - [ ] Fortran for fun and profit
 - [x] RxSwift by tutorials
-- [ ] Mastering git
+- [x] Mastering git
 - [ ] Care and feeding of developers
```

- That looks pretty obtuse, but a `diff` is simply a compact way of showing you what's changed between two files. In this case, Git is telling you that you're comparing two versions of the same file — the version of the file in your working directory, and the version of the file that you told Git to stage earlier with the `git add` command
- Add all the changes to the staging area with `git add .` That full stop (or period) character tells Git to add all changes to the staging area, both in this directory and all other subdirectories.
- Run `git diff` again. Uh, that's interesting. `git diff` reports that nothing has changed. But if you think about it for a moment, that makes sense. `git diff` compares your *working tree to the staging area*. With `git add .` , you put everything from your working tree into the staging area, so there should be no differences between your working tree and staging.
- If you want to be really thorough (or if you don't trust Git quite yet), you can ask Git to show you the differences that it's staged for commit and the working tree with an extra option on the end of git diff.



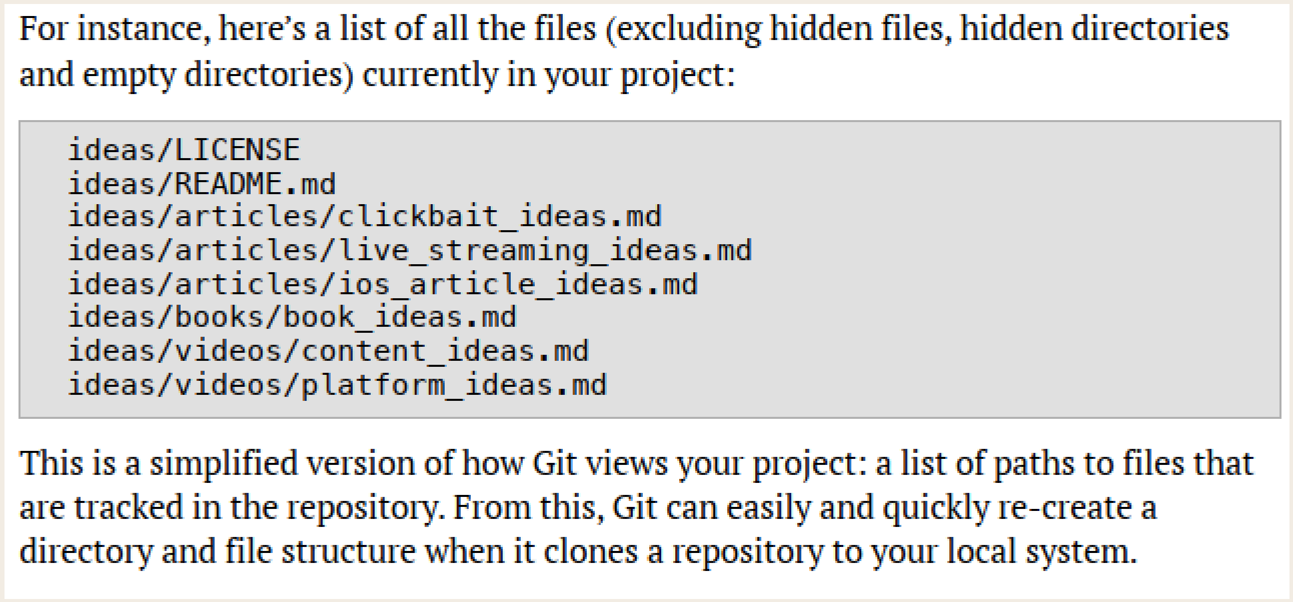
```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git diff

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git diff --staged
diff --git a/books/book_ideas.md b/books/book_ideas.md
index 1a92ca4..5086b1f 100644
--- a/books/book_ideas.md
+++ b/books/book_ideas.md
@@ -7,4 +7,5 @@
 - [ ] CVS by tutorials
 - [ ] Fortran for fun and profit
 - [x] RxSwift by tutorials
-- [ ] Mastering git
+- [x] Mastering git
+- [ ] Care and feeding of developers

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$
```

- Commit changes `git commit -m ""`

Adding directories

- create a new directory, `mkdir tutorials`
- check if the directory exists using `ls`
- It does. Now run `git diff`
- Er, that doesn't seem right. Why can't Git see your new directory? That's by design, and it reflects the way that Git thinks about files and directories.
- At its core, Git really only knows about files, and nothing about directories. Git thinks about a file as "a string that points to an entity that Git can track". If you think about this, it makes some sense: If a file can be uniquely referenced as the full path to the file, then tracking directories separately is quite

redundant.



For instance, here's a list of all the files (excluding hidden files, hidden directories and empty directories) currently in your project:

```
ideas/LICENSE
ideas/README.md
ideas/articles/clickbait_ideas.md
ideas/articles/live_streaming_ideas.md
ideas/articles/ios_article_ideas.md
ideas/books/book_ideas.md
ideas/videos/content_ideas.md
ideas/videos/platform_ideas.md
```

This is a simplified version of how Git views your project: a list of paths to files that are tracked in the repository. From this, Git can easily and quickly re-create a directory and file structure when it clones a repository to your local system.

- The solution to making Git recognize a directory is clearly to put a file inside of it. But what if you don't have anything yet to put here, or you want an empty directory to show up in everyone's clone of this project?
- **The solution is to use a placeholder file. The usual convention is to create a hidden, zero-byte .keep file inside the directory you want Git to "see."** Add this file using `touch tutorials/.keep`
- *The touch command was originally designed to set and modify the "modified" and "accessed" times of existing files. But one of the nice features of touch is that, if a specified file doesn't exist, touch will automatically create the file for you.*
- On running `git diff` now, you can see that git now recognizes the directory created.
- Add the changes to the staging area using `git add tutorials/\*` . While you could have just used `git add .` as before to add all files, this form of git add is a nice way to only add the files in a particular directory or subdirectory. In this case, you're telling Git to stage all files underneath the tutorials directory. Add a commit.

Looking at git log

- see the entire commit history, `git log`. (Use `Space` to navigate if there are a lot of commits, and `Esc` to exit.)
- To see every detail of your commits use, `git log -p`
- More on this in Git log and history

Key Points

- A commit is essentially a snapshot of the particular state of the set of files in the repository at a point in time.
- The working tree is the collection of project files that you work with directly.
- `git status` shows you the current state of your working tree.
- Git thinks about the files in your working tree as being in three distinct states: *unmodified*, *modified* and *staged*.
- `git add <filename>` lets you add changes from your working tree to the staging area.
- `git add .` adds all changes in the current directory and its subdirectories.
- `git add <directoryname>/\*` lets you add all changes in a specified directory.
- `git diff` shows you the difference between your working tree and the staging area.
- `git diff --staged` shows you the difference between your staging area and the last commit to the repository.
- `git commit` commits all changes in the staging area and opens Vim so you can add a commit message.
- `git commit -m "<your message here>"` commits your staged changes and includes a message without having to go through Vim.

- `git log` shows you the basic commit history of your repository.
- `git log -p` shows the commit history of your repository with the corresponding diffs.
- END

The Staging Area

Why staging exists

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.

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.

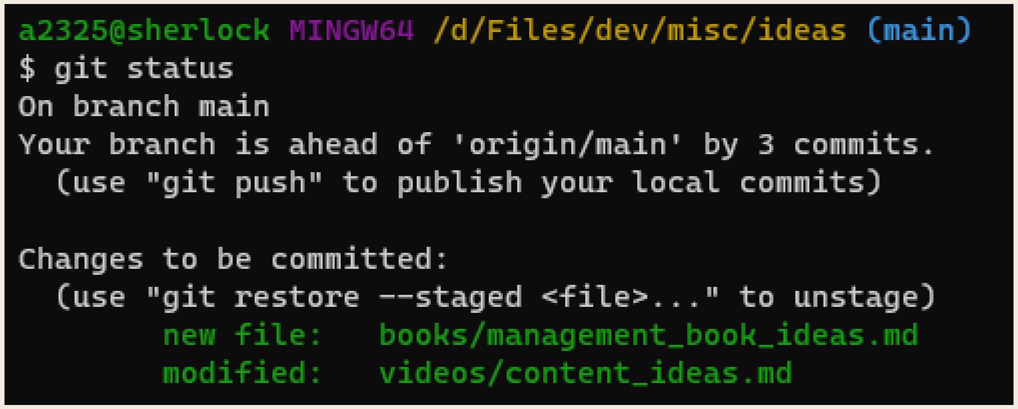
Git was built to compensate for this messy, non-linear approach to development.

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.”

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.

- Make the following additions
  - create a new file in the books directory, named management_book_ideas.md, `touch books/management_book_ideas.md`
  - 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? Open `notepad videos/content_ideas.md` and make the necessary changes
  - `git add .` to add all changes to staging area
  - Run `git status`



```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ 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)
        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?

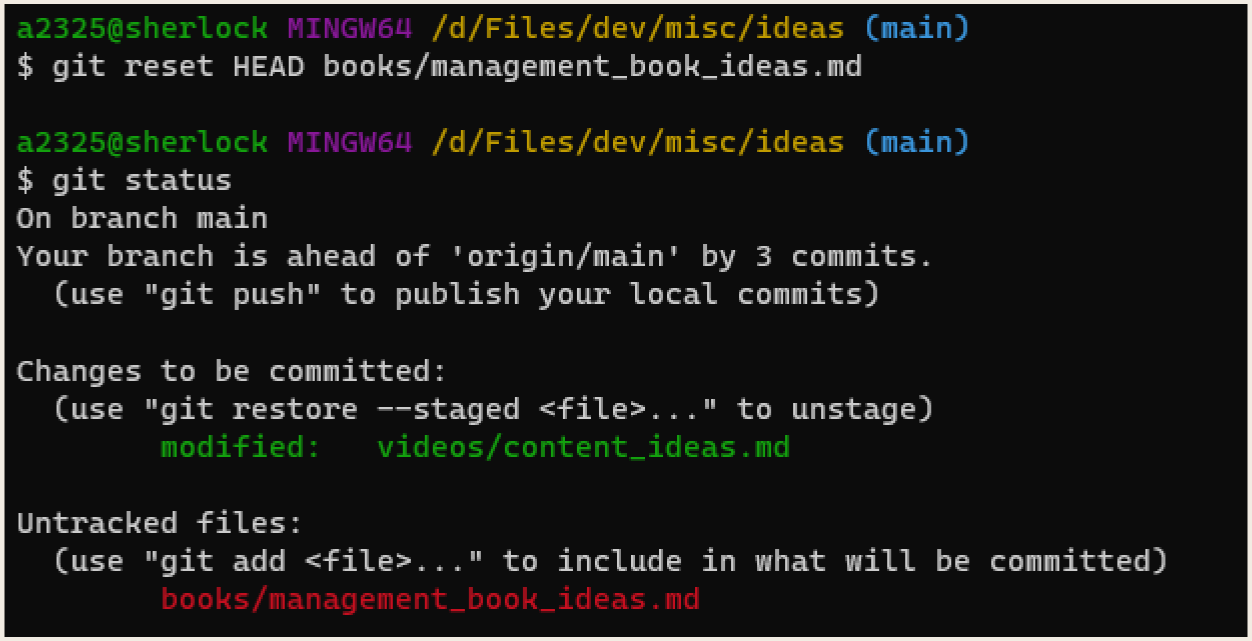
> - The easiest way to do this is through `git reset` .

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`

> - HEAD is simply a label that references the most recent commit.

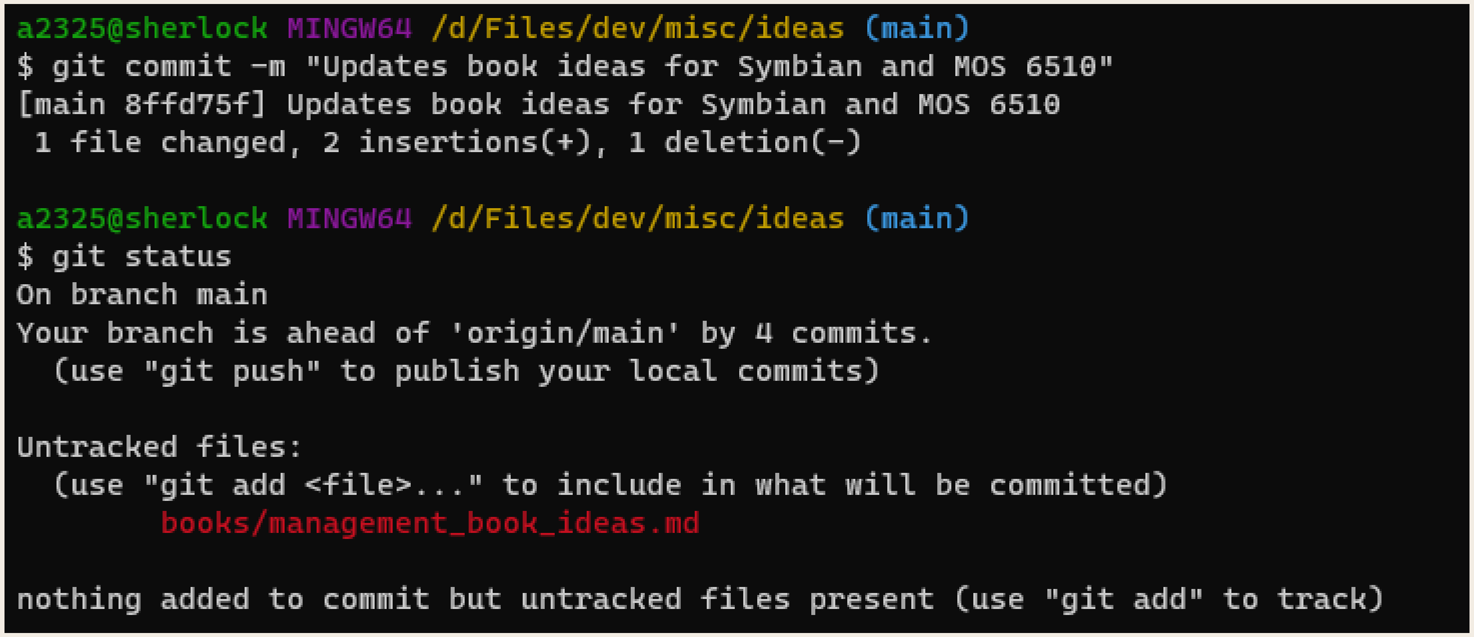
- So, `git reset HEAD books/management_book_ideas.md` , in this context means "use HEAD as a reference point, restore the staging area to that point, but only restore any changes related to the books/management_book_ideas.md file."
- Run the `git status` command



```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git reset HEAD books/management_book_ideas.md

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ 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
```

- Git is no longer tracking books/management_book_ideas.md, but it's still tracking your changes to videos/content_ideas.md

- Commit the changes `git commit -m "Updates book ideas for Symbian and MOS 6510"`

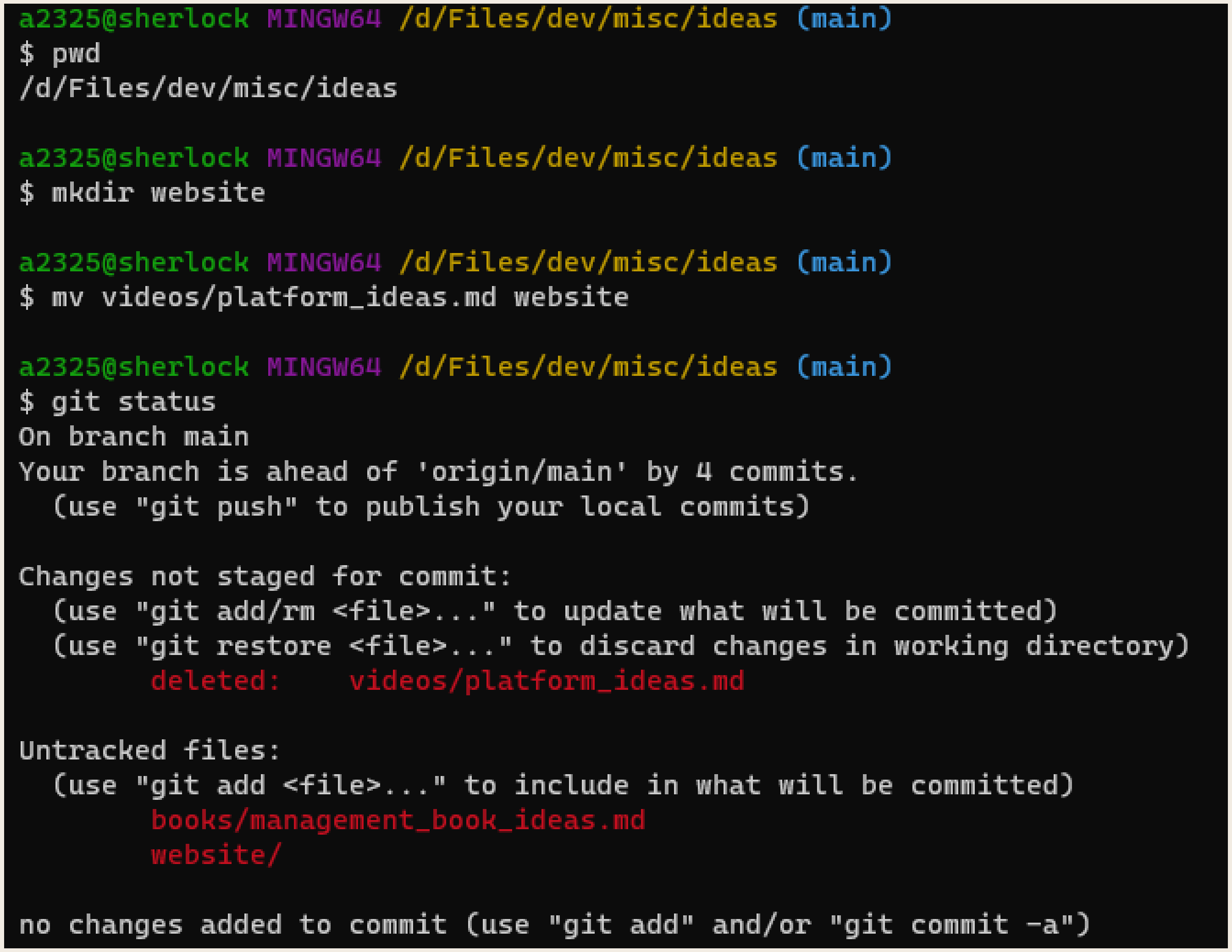


```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git commit -m "Updates book ideas for Symbian and MOS 6510"
[main 8ffd75f] Updates book ideas for Symbian and MOS 6510
 1 file changed, 2 insertions(+), 1 deletion(-)

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 4 commits.
  (use "git push" to publish your local commits)

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

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

Moving files in Git

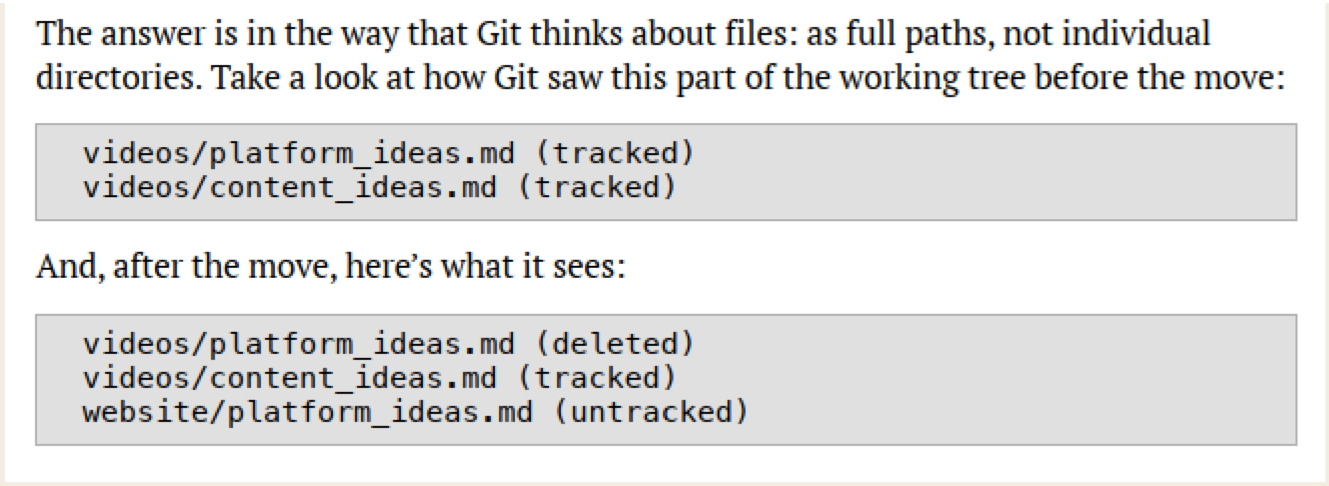
- create a new directory in the /ideas directory, `mkdir website`
- Move the file platform_ideas.md to this folder, do it manually using the mv command `mv videos/platform_ideas.md website`

- Execute `git status`

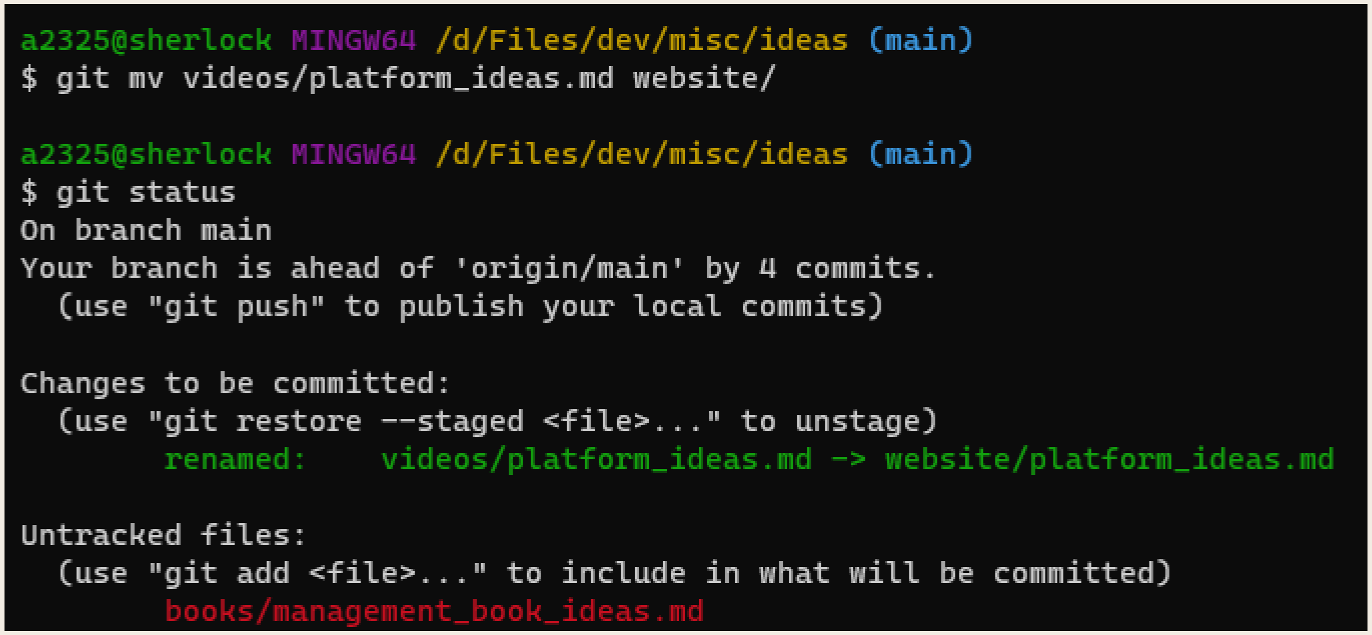


```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ pwd
/d/Files/dev/misc/ideas

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ mkdir website

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ mv videos/platform_ideas.md website

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ 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. Why doesn't it just see that you've moved the file?
    - Remember, Git knows nothing about directories: It only knows about full paths to files. Comparing the two snippets of your working tree below shows you exactly why git status reports what it does.


> 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)
> ```

- 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 to where it was, `mv website/platform_ideas.md videos/` . Git will go back to the state it was previously(before the move)
- Execute `git mv videos/platform_ideas.md website/` and then `git status`



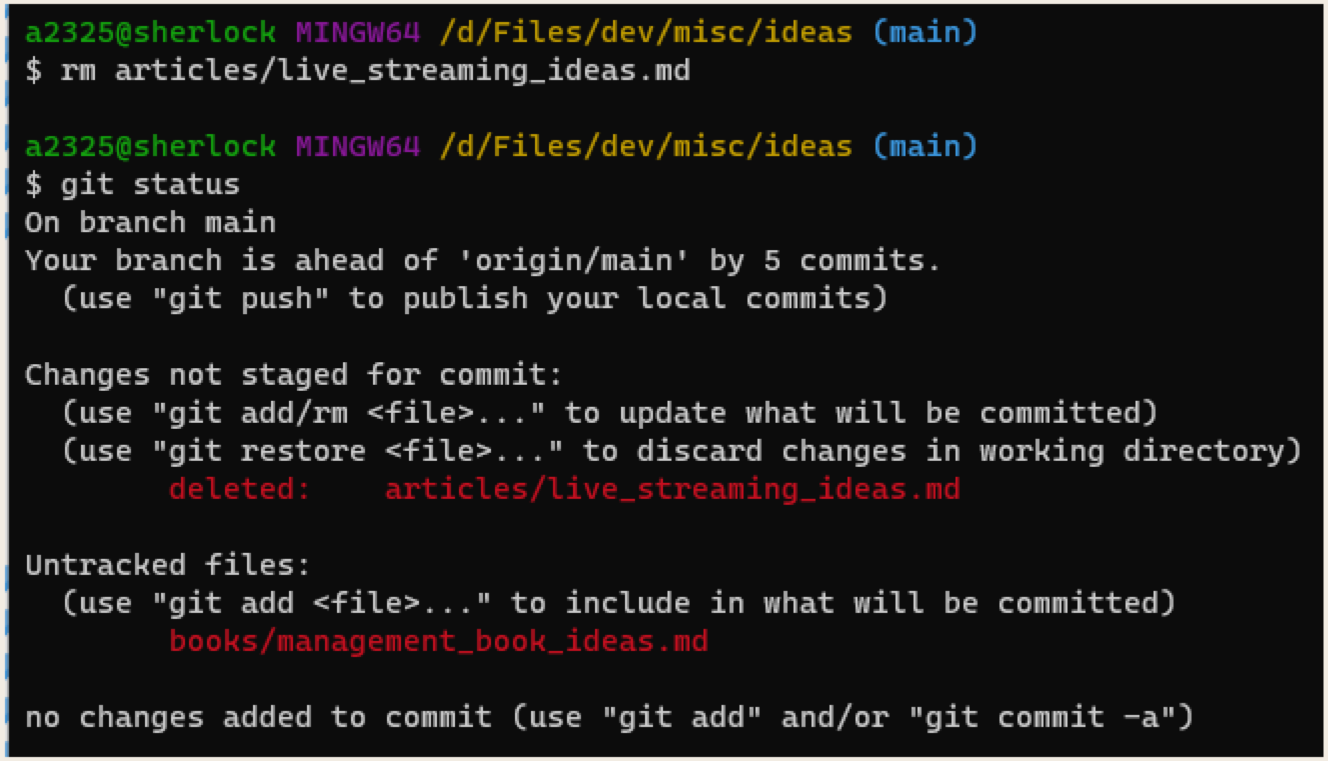
```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git mv videos/platform_ideas.md website/

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ 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
```

- 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.
- Commit the changes `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(articles/live_streaming_ideas.md) ideas are pretty bad. Perhaps you should just get rid of them now before too many people see them.

Deleting files in Git

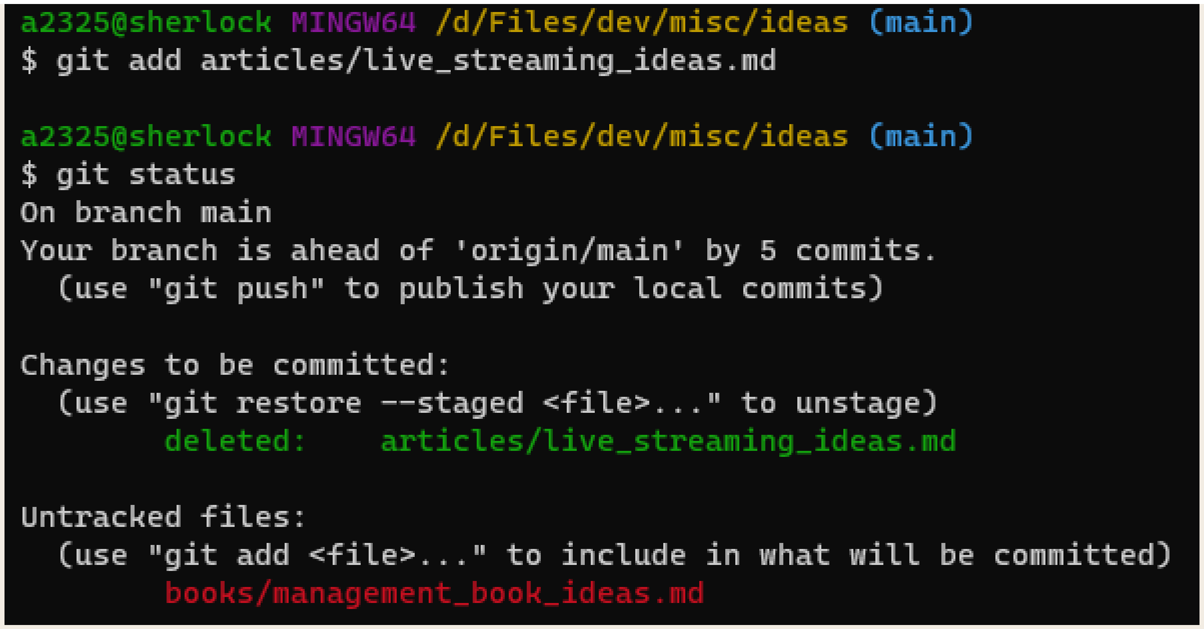
- The impulse to just delete/move/rename files as you’d normally do on your filesystem is usually what puts Git into a tizzy, and it causes people to say they don’t “get” Git. But if you take the time to instruct Git on what to do, it usually takes care of things quite nicely for you.
- So — that live streaming ideas file has to go.
- Brute force way

- Execute `rm articles/live_streaming_ideas.md` and then `git status`



```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ rm articles/live_streaming_ideas.md

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ 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")
```

- That's not so bad. Git recognizes that you've deleted the file and is prompting you to stage it. `git add articles/live_streaming_ideas.md`



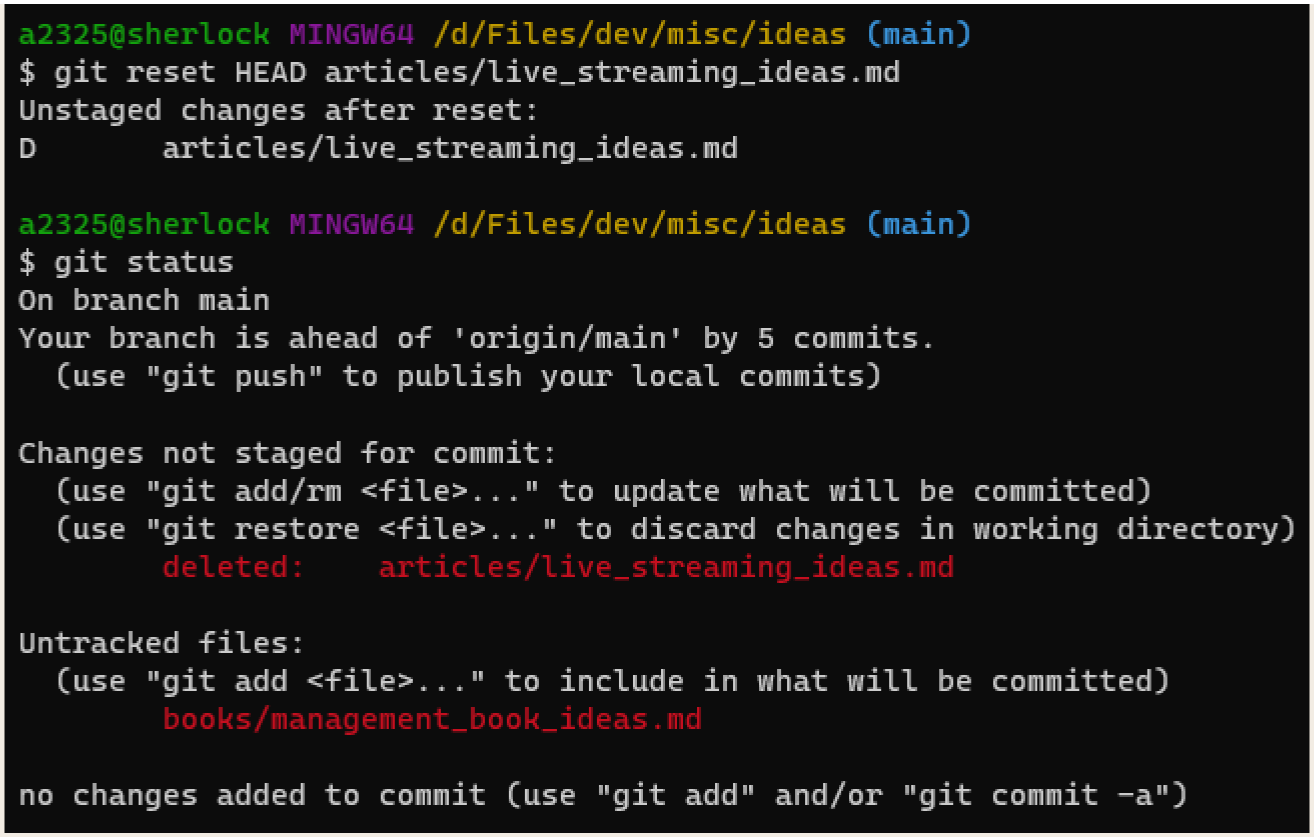
```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git add articles/live_streaming_ideas.md

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ 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.

- But to use `git rm` we have to go back to where we were

Restoring deleted files

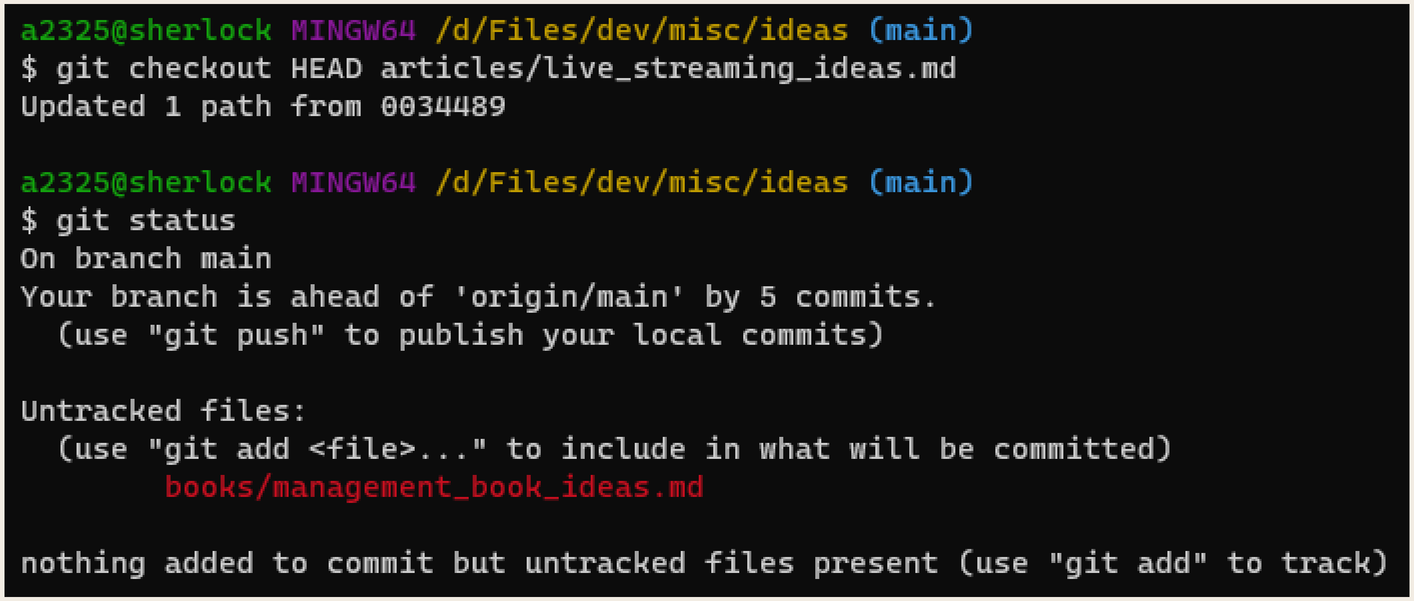
- Unstage the change to the live streaming ideas file `git reset HEAD articles/live_streaming_ideas.md`



```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git reset HEAD articles/live_streaming_ideas.md
Unstaged changes after reset:
D       articles/live_streaming_ideas.md

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ 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")
```

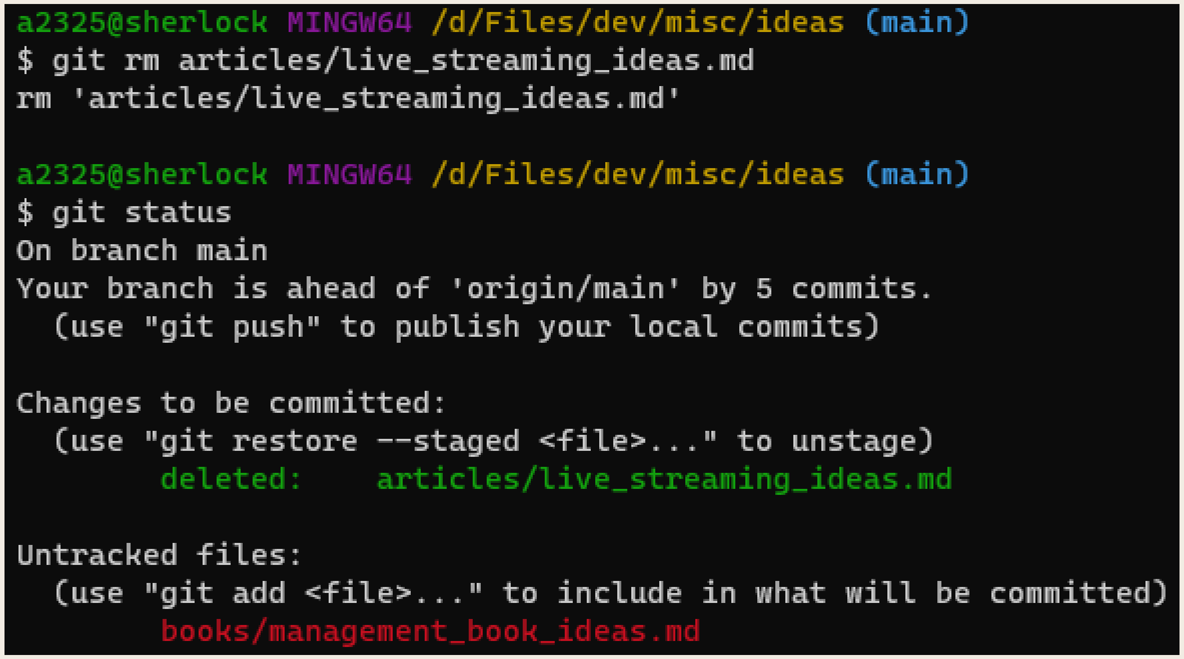
- That removes that change from the staging area — but it doesn't restore the file itself in your working tree. To do that, you'll need to tell Git to retrieve the latest committed version of that file from the repository.

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



```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git checkout HEAD articles/live_streaming_ideas.md
Updated 1 path from 0034489

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 5 commits.
  (use "git push" to publish your local commits)

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

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

- Get rid of that file with the following command `git rm articles/live_streaming_ideas.md`



```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git rm articles/live_streaming_ideas.md
rm 'articles/live_streaming_ideas.md'

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ 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
```

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

- add and commit the management_book_ideas.md file, `git add books/management_book_ideas.md` , `git commit -m "Adds all the good ideas about management"`

If you delete a file by mistake

- say the file is books/management_book_ideas.md
- To restore it

> - `git reset HEAD books/management_book_ideas.md`
> - `git checkout HEAD books/management_book_ideas.md`

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>`
- END

Ignoring Files in Ignoring files in Git

There are quite a few situations in which you might not want Git to track everything.

A good example would be any files that contain API keys, tokens, passwords or other secrets that you definitely need for testing, but you don't want them sitting in a repository — especially a public repository — for all to see.

`.gitignore` file, is a set of rules held in a file that tell Git to not track files or sets of files

But the real power of `.gitignore` is in its ability to pattern-match a wide range of files so that you don't have to spell out every single file you want Git to ignore.

You can have a global .gitignore that applies to all of your repositories, and then put project-specific .gitignore files within directories or subdirectories under the projects that need a particularly pedantic level of control.

Key points

- .gitignore lets you configure Git so that it ignores specific files or files that match a certain pattern.
- `*.html` in your .gitignore matches on all files with an .html extension, in any directory or subdirectory of your project.
- `*/*.html` matches all files with an .html extension, but only in subdirectories of your project.
- `!` negates a matching rule.
- You can have multiple .gitignore files inside various directories of your project to override higher-level matches in your project.
- You can find where your global .gitignore lives with the command `git config --global core.excludesfile`.
- GitHub hosts some excellent started .gitignore files at https://github.com/github/gitignore.

Git log and history

When you mess up your project (not if, but when), you'll want to be able to go back in history and find a commit that worked, and rewind your project back to that point in time. This chapter shows you how.

- `git log` This shows you a list of ancestral commits — that is, the set of commits that form the history of the current head
- `git log -3` shows the number of commits(last 3 here) you'd like to see, starting from the most recent
- `git log --oneline`
- The short hash is simply the first seven characters of the long hash
- `git log --graph` shows tree structure of your repository's history
- `git log --graph --oneline`
- Git's not showing you the complete history, though. It's only showing you the history of things that have happened on the master branch. To tell Git to show you the complete history of everything it knows about use the `--all`. `git log --oneline --graph --all`
- `git shortlog`, this is a nice way to get a summary of the commits

Searching Git history

- `git log --author=crispy8888 --oneline` search commits only from crispy8888 user. This does the same, `git log --author="Chris Belanger" --oneline`
- `git log --grep=ideas --oneline` find the commits, which have a commit message that contains the word "ideas"
- Execute the following command to see all of the full commit messages for books/book_ideas.md. `git log --oneline books/book_ideas.md`

- You can also see the commits that happened to the files in a particular directory, `git log --oneline books` . This shows you all the changes that happened in that directory, but it's not clear which files were changed. `git log --oneline --stat books` , To get a clearer picture of which files were changed in that directory, you can throw the `--stat` option
- Find all of the commits in your code that deal with the term "Fortran", `git log -S"Fortran"`
- `git log -S"Fortran" -p`

Key points

- `git log` by itself shows a basic, vanilla view of the ancestral commits of the current HEAD.
- `git log -p` shows the diff of a commit.
- `git log -n` shows the last n commits.
- `git log --oneline` shows a concise view of the short hash and the commit message.
- You can stack options on git log, as in `git log -8 --oneline` to show the last 8 commits in a condensed form.
- `git log --graph` shows a crude but workable graphical representation of your repository.
- `git log --all` shows commits on other branches in the repository, not just the ancestors of the current HEAD.
- `git shortlog` shows a summary of commits, grouped by their author them, in increasing time order.
- `git log --author="<authorname>"` lets you search for commits by a particular author.
- `git log --grep="<term>"` lets you search commit messages for a particular term.
- `git log <path/to/filename>` will show you just the commits associated with that one file.
- `git log <directory>` will show you the commits for files in a particular directory.
- `git log --stat` shows a nice overview of the scope and scale of the change in each commit.
- `git log -S"<term>"` lets you search the contents of a commit's changeset for a particular term.

What is a commit?

- You probably think about your files primarily in terms of their content, their position inside the directory hierarchy, and their names. So when you think of a commit, you're likely to think about the state of the files, their content and names at a particular point in time. And that's correct, to a point
- Git also adds some more information to that "state of your files" concept in the form of metadata
- Git metadata includes such things like "when was this committed?" and "who committed this?", but most importantly, it includes the concept of "where did this commit originate from?" — and that piece of information is known as the commit's parent.
- Git takes all that metadata, including a reference to this commit's parent, and wraps that up with the state of your files as the commit. Git then hashes that collection of things using SHA1 to create an ID, or key, that is unique to that commit inside your repository.

What is a branch?

**It's simply a reference, or a label, to a commit in your repository.

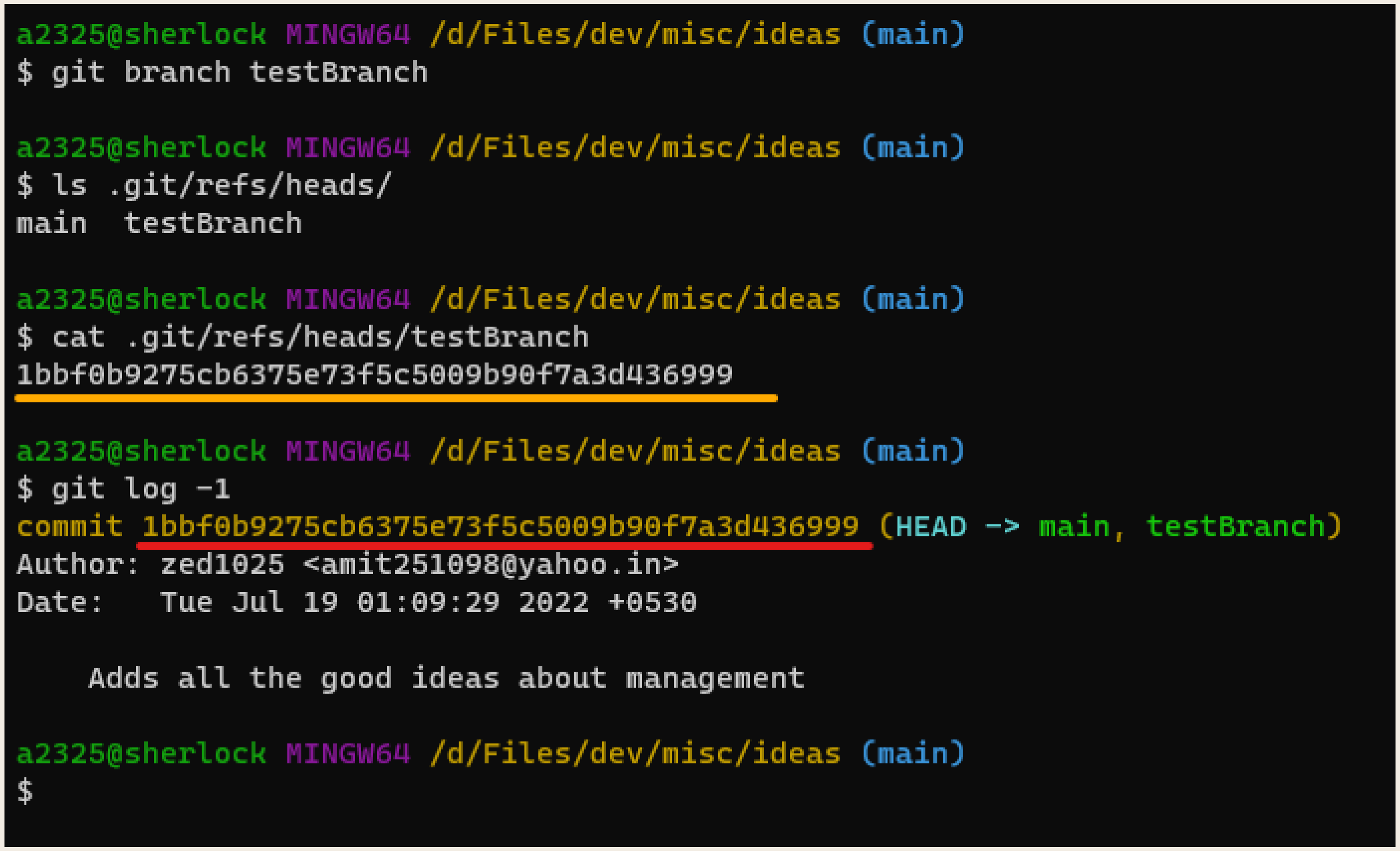
- And because you can refer to a commit in Git simply through its hash, you can see how creating branches is a terribly cheap operation. There's no copying, no extra cloning, just Git saying "OK, your new branch is a label to commit 477e542". Boom, done.
- As you make commits on your branch, that label for the branch gets moved forward and updated with the hash of each new commit. Again, all Git does is update that label, which is stored as a simple file

in that hidden .git repository, as a really cheap operation.

- `git config --global init.defaultBranch main` . This sets the default branch name to main. This only affects new repositories that you create; it doesn't change the default branch name of any existing repositories.

How Git tracks branches

- `git branch testBranch` creates a new branch

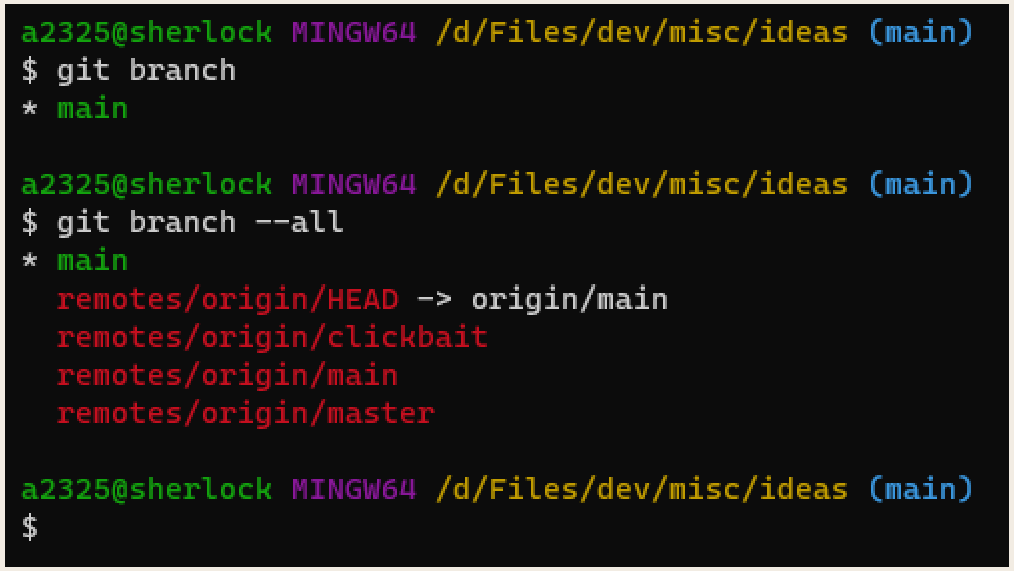


```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git branch testBranch

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ ls .git/refs/heads/
main  testBranch

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ cat .git/refs/heads/testBranch
1bbf0b9275cb6375e73f5c5009b90f7a3d436999

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git log -1
commit 1bbf0b9275cb6375e73f5c5009b90f7a3d436999 (HEAD -> main, testBranch)
Author: zed1025 <amit251098@yahoo.in>
Date:   Tue Jul 19 01:09:29 2022 +0530

    Adds all the good ideas about management

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$
```

- .git/refs/heads/ directory contains the files that point to all of your branches.

Switching to another branch

- `git branch` by itself only shows the local branches in your repository



```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git branch
* main

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$ git branch --all
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/clickbait
  remotes/origin/main
  remotes/origin/master

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (main)
$
```

- the remote only has one branch: `clickbait`
- To get this branch down to your machine, tell Git to start tracking it, and switch to this branch all in one action, execute the following command `git checkout --track origin/clickbait`

Explaining origin

origin is another one of those convenience conventions that Git uses. Just like master is the default name for the first branch created in your repository, *origin is the default alias for the location of the remote repository from where you cloned your local repository*.

- execute the following command to see where Git thinks origin lives `git remote -v`
- To see Git's view of all local and remote branches now, execute the following command, `git branch --all -v`

A shortcut for branch creation

- `git checkout --track origin/clickbait` can be replaced with `git checkout clickbait`. When you specify a branch name to git checkout, Git checks to see if there is a local branch that matches that name to switch to. If not, then it looks to the origin remote, and if it finds a branch on the remote matching that name, it assumes that is the branch you want, checks it out for you, and switches you to that branch.
- There's also a shortcut command which solves the two-step problem of `git branch <branchname>` and `git checkout <branchname>`: `git checkout -b <branchname>`.

Key points

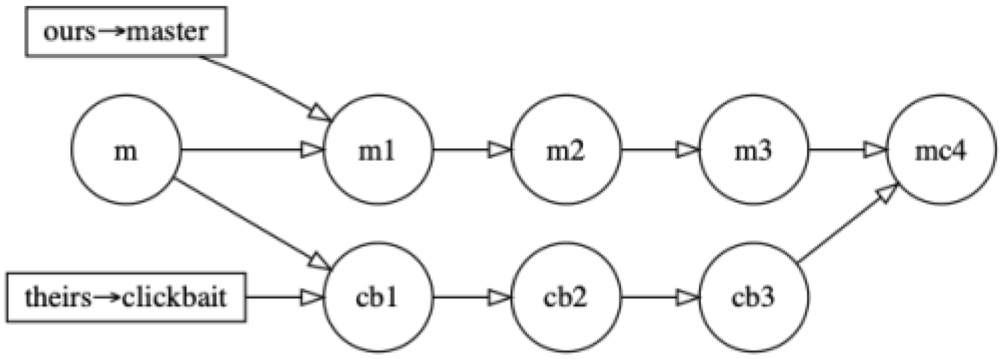
- A commit in Git includes information about the state of the files in your repository, along with metadata such as the commit time, the commit creator, and the commit's parent or parents.
- The hash of your commit becomes the unique ID, or key, to identify that particular commit in your repository.
- A branch in Git is simply a reference to a particular commit by way of its hash.
- `master` is simply a convenience convention, but has come to be accepted as the original branch of a repository. `main` is also another common convenience branch name in lieu of master.
- Use `git branch <branchname>` to create a branch.
- Use `git branch` to see all local branches.
- Use `git checkout <branchname>` to switch to a local branch, or to checkout and track a remote branch.
- Use `git branch -d <branchname>` to delete a local branch.
- Use `git branch --all` to see all local and remote branches.

- `origin` , like master, is simply a convenience convention that is an alias for the URL of the remote repository.
- Use `git checkout -b <branchname>` to create and switch to a local branch in one fell swoop.

Merging

- Git thinks about branches in terms of ours and theirs. “Ours” refers to the branch to which you’re merging back to, and “theirs” refers to the branch that you want to pull into “ours”.

Let’s say you want to merge the `clickbait` branch back into `master`. In this case, as shown in the diagram below, `master` is **ours** and the `clickbait` branch would be **theirs**. Keeping this distinction straight will help you immeasurably in your merging career.

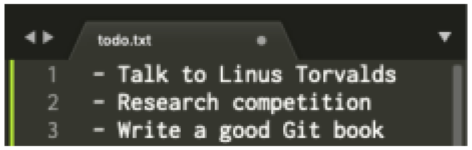


$$\begin{tikzcd}
\boxed{\text{ours}\rightarrow\text{master}} \arrow[dr] & & & & \\
\text{m} \arrow[r] \arrow[dr] & \text{m1} \arrow[r] & \text{m2} \arrow[r] & \text{m3} \arrow[r] & \text{mc4} \\
\boxed{\text{theirs}\rightarrow\text{clickbait}} \arrow[r] & \text{cb1} \arrow[r] & \text{cb2} \arrow[r] & \text{cb3} \arrow[ur] &
\end{tikzcd}$$

Three-way merges

- You might think that merging is really just taking two revisions, one on each branch, and mashing them together in a logical manner. This would be a two-way merge. However, a merge in Git actually uses three revisions to perform what is known as a three-way merge.

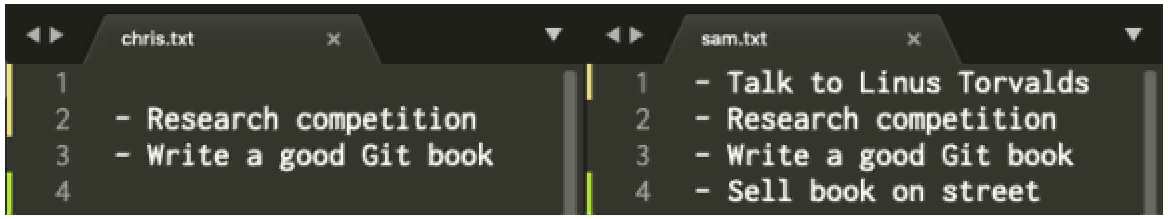
To see why this is, take a look at the two-way merge scenario below. You have one simple text file; you're working on one copy of the file while your friend is working on another, separate copy of that same file.



```
- Talk to Linus Torvalds
- Research competition
- Write a good Git book
```

The original file.

You delete a line from the top of the file, and your friend adds a line to the bottom of the file.

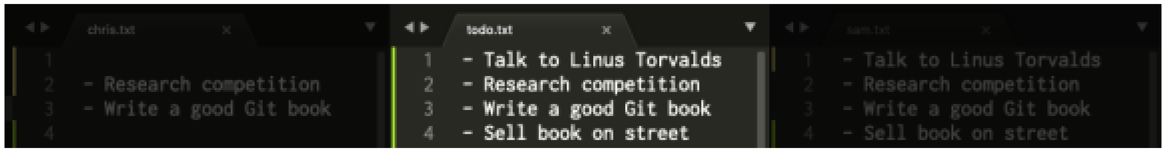


```

- Research competition
- Write a good Git book

```

```
- Talk to Linus Torvalds
- Research competition
- Write a good Git book
- Sell book on street
```

Chris' changes on the left; Sam's changes on the right.

Now imagine that you and your friend hand off your work to an impartial third party to merge this text file together. Now, this third party has literally no idea as to what the original state of this file was, so she has to make a guess as to what she should take from each file.



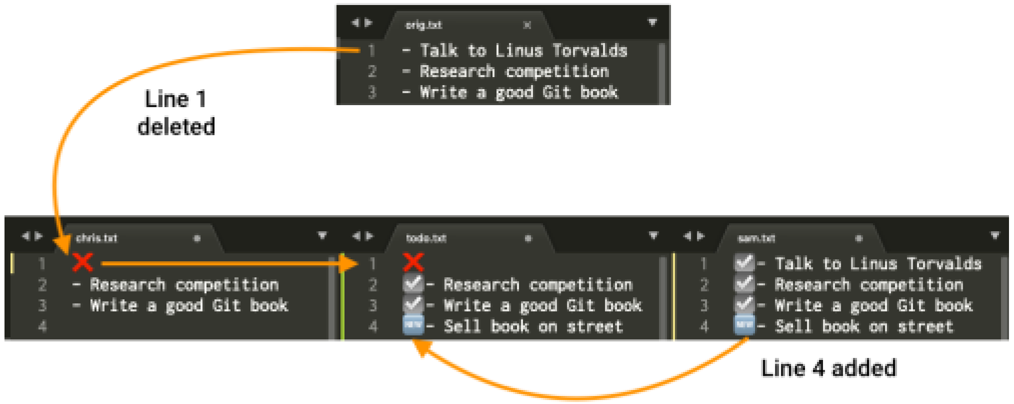
```

- Research competition
- Write a good Git book

```

```
- Talk to Linus Torvalds
- Research competition
- Write a good Git book
- Sell book on street
```

```
- Talk to Linus Torvalds
- Research competition
- Write a good Git book
- Sell book on street
```

With no background of what the starting point was, the person responsible to merge tries to preserve as many lines as possible in common to both files.

Now, imagine you and your friend *also* provided the original file that you both started with — the common ancestor — to your impartial third party. She could compare each new file's changes to the original file, figure out the diff of your changes, figure out the diff of your friend's changes, and create the correct resulting merged document from the diffs of each.



```
orig.txt
- Talk to Linus Torvalds
- Research competition
- Write a good Git book
```

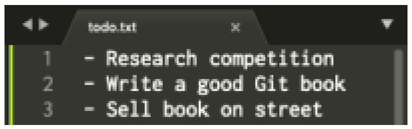
```
chris.txt
- Research competition
- Write a good Git book
```

```
todo.txt
- Research competition
- Write a good Git book
- Sell book on street
```

```
sam.txt
- Talk to Linus Torvalds
- Research competition
- Write a good Git book
- Sell book on street
```

Knowing the origin of each set of changes lets you detect that Line 1 was deleted by Chris, and Line 4 was added by Sam.

That's better. And this, essentially, is what Git does in an automated fashion. By performing three-way merges on your content, Git gets it right most of the time. Once in a while, Git won't be able to figure things out on its own, and you'll have to go in there and help it out a little bit. But you'll get into these scenarios a little later on in this book when you work on **merge conflicts**, which are a lot less scary than they sound.



```
todo.txt
- Research competition
- Write a good Git book
- Sell book on street
```

The result is what you both intended.

Merging a branch

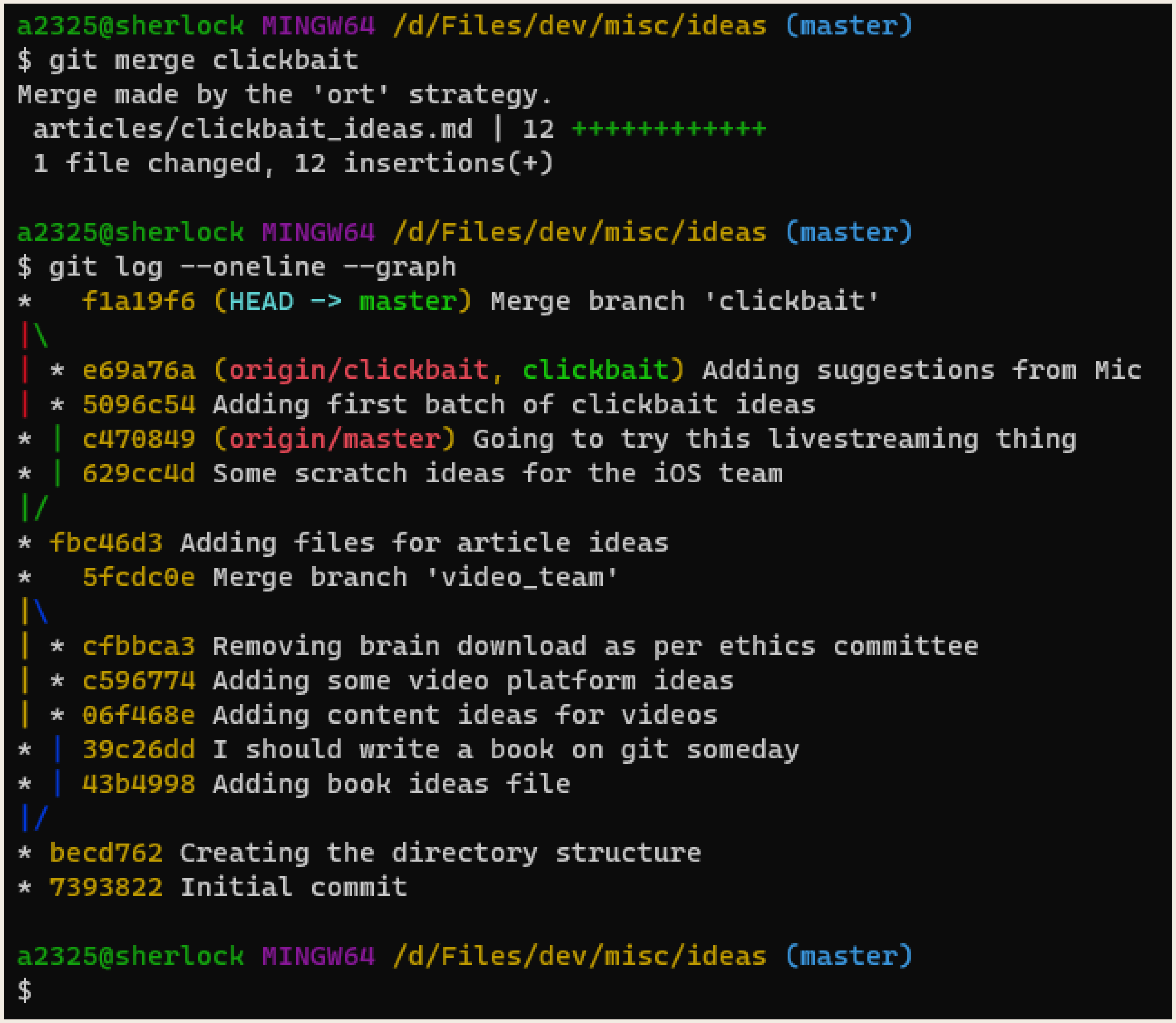
- We will merge changes on the clickbait branch to the master
- `git checkout clickbait`

- Execute the following command to see what's been committed on this branch that you'll want to merge back to master, `git log clickbait --not master`

    > - This little gem is quite nice to keep on hand, as it tells you "what are the commits that are just in the clickbait branch, but not in master?"

- Recall that merging is the action of pulling in changes that have been done on another branch. In this case, you want to pull the changes from clickbait into the master branch. To do that, you'll have to be on the master branch first.
- move to master `git checkout master`

- merge using `git merge clickbait`



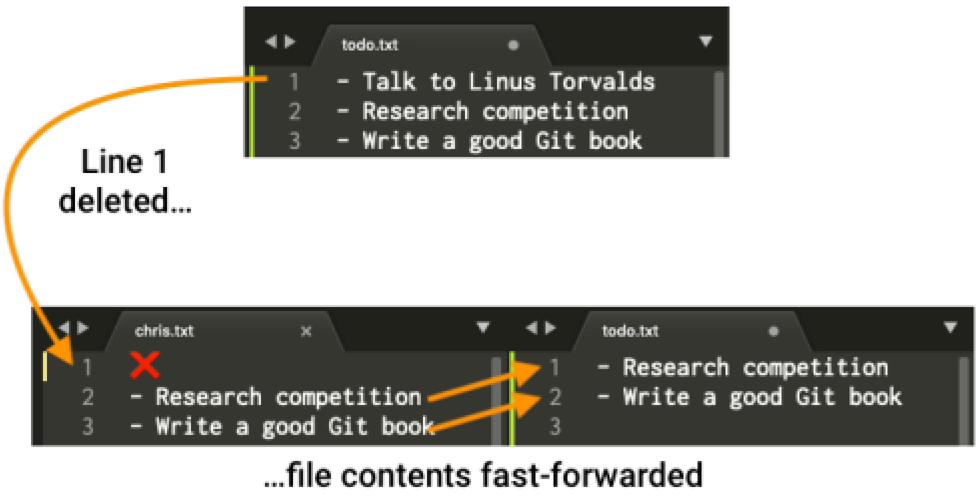
```
a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (master)
$ git merge clickbait
Merge made by the 'ort' strategy.
 articles/clickbait_ideas.md | 12 ++++++++++++
 1 file changed, 12 insertions(+)

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (master)
$ git log --oneline --graph
*   f1a19f6 (HEAD -> master) Merge branch 'clickbait'
|\
| * e69a76a (origin/clickbait, clickbait) Adding suggestions from Mic
| * 5096c54 Adding first batch of clickbait ideas
* | c470849 (origin/master) Going to try this livestreaming thing
* | 629cc4d Some scratch ideas for the iOS team
|/
* fbc46d3 Adding files for article ideas
*   5fcdc0e Merge branch 'video_team'
|\
| * cfbbca3 Removing brain download as per ethics committee
| * c596774 Adding some video platform ideas
| * 06f468e Adding content ideas for videos
* | 39c26dd I should write a book on git someday
* | 43b4998 Adding book ideas file
|/
* becd762 Creating the directory structure
* 7393822 Initial commit

a2325@sherlock MINGW64 /d/Files/dev/misc/ideas (master)
$
```

Fast-forward merge

There's another type of merge that happens in Git, known as the **fast-forward** merge. To illustrate this, think back to the example above, where you and your friend were working on a file. Your friend has gone away (probably hired away by Google or Apple, lucky sod), and you're now working on that file by yourself.

Once you've finished your revisions, you take your updated file, along with the original file (the common ancestor, again) to your impartial third party for merging. She's going to look at the common ancestor file, along with your new file, but she isn't going to see a third file to merge.

In this case, she's just going to commit your file on top of of the old file, because *there's nothing to merge*.



```
- Talk to Linus Torvalds
- Research competition
- Write a good Git book
```

```

- Research competition
- Write a good Git book
```

```
- Research competition
- Write a good Git book
```

If there are no other changes to the file to merge, Git simply commits your file over top of the original.

If no other person had touched the original file since you picked it up and started working on it, there's no real point in doing anything fancy, here. And while Git is far from lazy, it is terribly efficient and only does the work it absolutely needs to do to get the job done. This, in effect, is exactly what a fast-forward merge does.

To see this in action, you'll create a branch off of master, make a commit, and then merge the branch back to master to see how a fast-forward merge works.

First, execute the following to ensure you're on the `master` branch:

```
git checkout master
```

Now, create a branch named `readme-updates` to hold some changes to the README.md file:

```
git checkout -b readme-updates
```

Git creates that branch and automatically switches you to it. Now, open **README.md** in your favorite text editor, and add the following text to the end of the file:

```
This repository is a collection of ideas for articles, content
and features at raywenderlich.com.

Feel free to add ideas and mark taken ideas as "done".
```

Save your changes, and return to Terminal. Stage your changes with the following command:

```
git add README.md
```

Now, commit that staged change with an appropriate message:

```
git commit -m "Adding more detail to the README file"
```

Now, to merge that change back to `master`. Remember — you need to be on the branch you want to pull the changes *into*, so you'll have to switch back to `master` first:

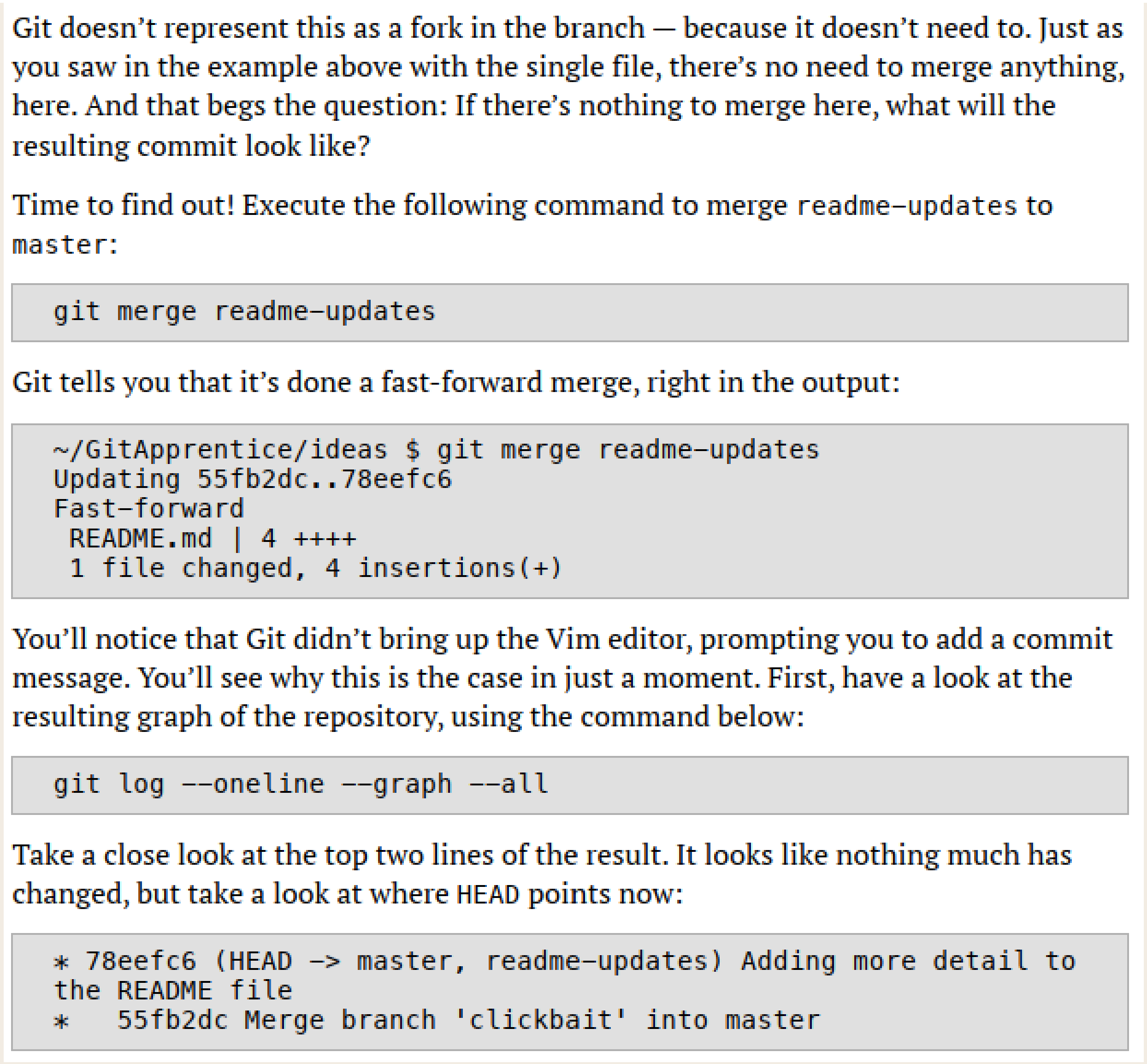
```
git checkout master
```

Now, before you merge that change in, take a look at Git's graph of the repository, using the `--all` flag to look on all branches, not just `master`:

```
git log --oneline --graph --all
```

Take a look at the top two lines of the result:

```
* 78eefc6 (readme-updates) Adding more detail to the README file
*   55fb2dc (HEAD -> master) Merge branch 'clickbait' into
master
```


Git doesn’t represent this as a fork in the branch — because it doesn’t need to. Just as you saw in the example above with the single file, there’s no need to merge anything, here. And that begs the question: If there’s nothing to merge here, what will the resulting commit look like?

Time to find out! Execute the following command to merge `readme-updates` to `master`:

```
git merge readme-updates
```

Git tells you that it’s done a fast-forward merge, right in the output:

```
~/GitApprentice/ideas $ git merge readme-updates
Updating 55fb2dc..78eefc6
Fast-forward
 README.md | 4 ++++
 1 file changed, 4 insertions(+)
```

You’ll notice that Git didn’t bring up the Vim editor, prompting you to add a commit message. You’ll see why this is the case in just a moment. First, have a look at the resulting graph of the repository, using the command below:

```
git log --oneline --graph --all
```

Take a close look at the top two lines of the result. It looks like nothing much has changed, but take a look at where `HEAD` points now:

```
* 78eefc6 (HEAD -> master, readme-updates) Adding more detail to
the README file
*   55fb2dc Merge branch 'clickbait' into master
```

Here, all Git has done is move the HEAD label to your latest commit. And this makes sense; Git isn’t going to create a new commit if it doesn’t have to. It’s easier to just move the HEAD label along, since there’s nothing to merge in this case. And that’s why Git didn’t prompt you to enter a commit message in Vim for this fast-forward merge.

Forcing merge commits

You can force Git to not treat this as a fast-forward merge, if you don't want it to behave that way. For instance, you may be following a particular workflow in which you check that certain branches have been merged back to master before you build.

But if those branches resulted in a fast-forward merge, for all intents and purposes, it will look like those changes were done directly on master, which isn't the case.

To force Git to create a merge commit when it doesn't really need to, all you need to do is add the `--no-ff` option to the end of your merge command. The challenge for this chapter will let you create a fast-forward situation, and see the difference between a merge commit and a fast-forward merge.

Key Points

- Merging combines work done on one branch with work done on another branch.
- Git performs three-way merges to combine content.
- Ours refers to the branch to which you want to pull changes into; theirs refers to the branch that has the changes you want to pull into ours.
- `git log <theirs> --not <ours>` shows you what commits are on the branch you want to merge, that aren't in your branch already.
- `git merge <theirs>` merges the commits on the "theirs" branch into "our" branch.
- Git automatically creates a merge commit message for you, and lets you edit it before continuing with the merge.
- A fast-forward merge happens when there have been no changes to "ours" since you branched off "theirs", and results in no merge commit being made.
- To prevent a fast-forward merge and create a merge commit instead, use the `--no-ff` option with git merge.