Git

Jonathan Hodgson (Archie)

June 17, 2020

A few people recently have asked me about Git.

Git has become the de-facto for most situations.

Microsoft recently moved to git for version controlling Windows and Office.

I will not be able to cover everything relating to Git, it is an incredibly powerful tool. However, hopefully I will be able to give you enough to get started and at least understand the official documentation.

Aims

I am obviously not going to be able to go over everything that git does.

- I don't know everything Git does
- Git does LOADS of stuff

Hopefully after this you will be able to use Git well for most day-to-day tasks. Git has very compressive documentation. I hope that this will also give you enough of a background to understand the documentation.

What is Git

A very versatile Version Control System

- Keep track of source code (or other folders and files) and its history
- Facilitate collaboration
- Distributed

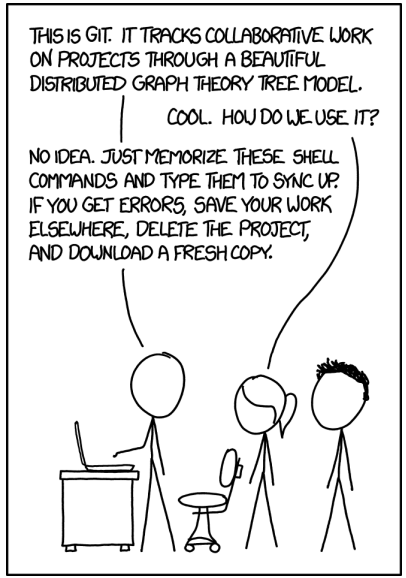
Git is still being developed.

Being distributed means you can work on repositories offline (Unlike SVN).

It's useful even if you're working on things by your self. This presentation is version controlled.

You can use it to find out when something broke. I won't be covering it today but there is a tool called git bisect that can take a unit test (or script) to analyse when something broke using a binary search.

Obligitary XKCD Comic



I have done this

Git has a reputation for being hard.

It's interface abstracts away a lot of the work, meaning it's commands can feel like magic. When it works, this is fine but unfortunately when things go wrong, you can be left - like in this comic - with no idea how to proceed.

It's interface can be confusing, there are some commands that do a lot (checkout) and there are often multiple ways to achieve something.

I think that understanding a bit about how Git works under the hood will help de-mistily it.

Git's data model is actually quite simple (beautiful even). Understanding the basics of this can really help.

Install

```
# Ubuntu / Debian / Kali
sudo apt install git

# Centos / Fedora / Red Hat
sudo dnf install git

# Arch / Antergos / Manjaro
sudo pacman -S git

# Mac
brew install git

# Get the Version
git --version
```

Git for Windows: https://gitforwindows.org/

Git is probably already installed if you are on a Linux system. However, if not, it will definitely be in your standard repositories.

There is a version of Git provided with xcode, but it is old. Most of the stuff we cover today should still work but (for example) some things need to be run from the root directory in old versions of git that don't in newer versions.

If you have the misfortune to be using windows, I've heard good things about Git for Windows but have not used it personally. It includes Bash emulation.

Hopefully you have a version greater than 2.23.0 - if not, it's not the end of the world.

Setting It Up

User

```
git config --global user.name "Jonathan Hodgson"
git config --global user.email "git@jonathanh.co.uk"
```

Hopefully you have Git installed. I will be running it on Linux although the commands should all be the same for Windows and Mac.

Note that I am not using my primary email address. The email address you provide here will be available to anyone with access to repositories you work on.

These settings are stored in `~/.gitconfig`.

Setting It Up

Editor

Pick One

```
# Set editor to vim
git config --global core.editor "vim"

# Set editor to nano
git config --global core.editor "nano"

# Set editor to VS Code
git config --global core.editor "code -w"

# Set editor to Sublime
git config --global core.editor "subl -w"
```

There are several times that Git will need to open a text editor. By default, it will use `EDITOR`. If neither is set, it will use VI.

Note that if you are using a GUI editor, you might have to set the wait flag. This makes it so the executable doesn't return until you close it.

Terminology

Objects

Blob In Git, a file is called a blob.

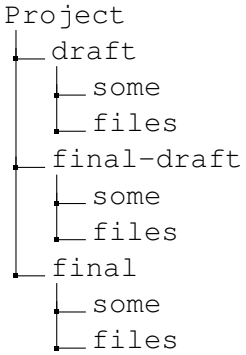
Tree In Git, a directory is called a tree.

Commit A snapshot of your code

Technically, a blob is kind of like an inode on the file system so also represents symbolic links.

There are some other types such as submodules but I won't be addressing them in this presentation.

Naïve Approach



Pros

- Simple
- No dependencies
- No Learning curve

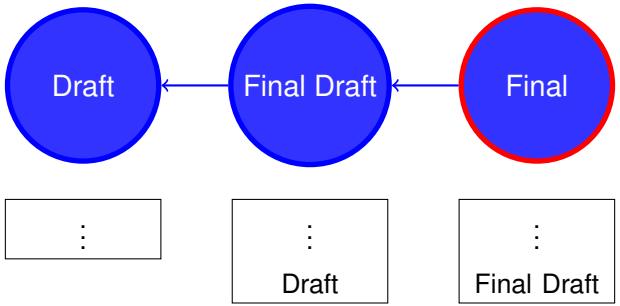
Cons

- Difficult to collaborate
- Lot's of wasted disk space
- Can be difficult to work out chronological order

I think, being honest, we have all done this. This sort of works, if you're working on something by yourself. Once you start collaborating on software, you are going to have a bad time.

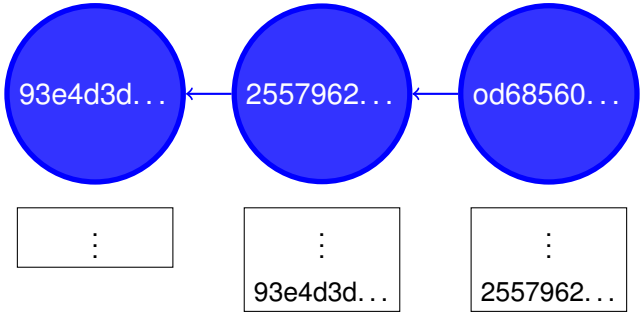
However, this is a simple approach and not a million miles from what Git does internally.

Model it



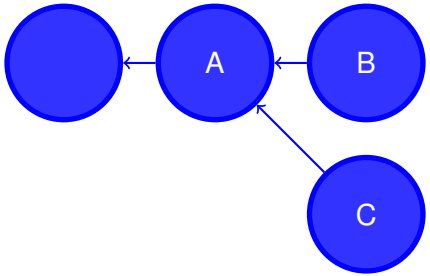
- This is a simple representation of the folder structure we saw.
- Notice that so the computer knows the order, somewhere in each “snapshot”, we include a reference to the previous snapshot.
- We then just need to record the most recent version somewhere.

Commits



- Rather than human readable names, Git references each snapshot (called a commit) by a cryptographic hash. Currently using a hardened sha1 but there is an effort to move to sha256.
- Similarly to the model above, each commit references the previous (except the first obviously)
- The commit also includes meta information such as the committer, a timestamp and a message.
- We will look at this in more detail a bit later.

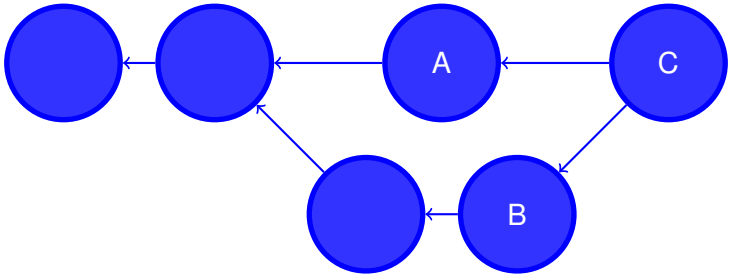
Commits / Branches



The linear graph we just saw is an overly simplistic representation. In reality, Git represents history using a Directed acyclic graph which allows parents to be shared my multiple commits. This is useful because it allows for Branches. We will look at these a bit more later.

It is good practice to develop features on a separate branch. This allows for multiple people to work on a project as well as allowing things like bug-fixes to be deployed without having to worry about interference from a new feature.

Commits / Branches



As well as 2 commits' ability to share a parent, the opposite is also true, Here, we see that a commit is able to have multiple parents.

This is called a merge commit - because it merges two branches. In a lot of situations git is smart enough to auto-merge branches although at times human intervention is necessary.

By default, git creates a branch called Master when you create a repository.

Create a repository

```
mkdir /tmp/demo
cd /tmp/demo
git init
Initialized empty Git repository in /tmp/demo/.git/
git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Do this in a live terminal. MAKE SURE YOU MAKE YOUR FONT BIGGER

Show that the `.git` folder has been created and do a tree to show what is in it.

Git status

```
touch greeting.py
chmod +x !$
vim greeting.py
git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    greeting.py

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

Create repo and create a file called greeting.py. Make sure to mark it as executable.

Here we see the branch we are on (Master), we are told that there are no commits yet and we see that Git can see the file we've just made but it isn't tracking it.

Staging Area

```
# Add files / or directories
git add <file|directory> [<file|directory>...]
# Add everything not in gitignore
git add -A
```

The staging area is where you put things that you want to be committed.

It can often be useful to manually split changes up into different commits. You might be working on feature A and feature B simultaneously. It is good practice to have each feature as a separate commit so you could add feature A to the staging area, commit it, then do the same for feature B.

We will talk about `.gitignore` in a bit.

Staging Area

```
❱ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    greeting.py

nothing added to commit but untracked files present (use "git add" to track)
❱ git add greeting.py
❱ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file: greeting.py
```

Here can use git status to see what is in the staging area. They are listed in the “Changes to be committed” section. By default, they will also be green if you have colour switched on.

Committing

```
git commit
```

- First line should be concise summary around 50 chars
- Body Should be wrapped to around 70 chars
- There should be an empty line separating summary from body
- If contributing to a project, check per-project guidelines
  - Normally in contributing.md or similar
- Use the imperative: “Fix bug” and not “Fixed bug” or “Fixes bug.”

First line is often shown by various tools

70 chars allows for good email etiquette. Allowing for 80 char hard wrap with after a few reply indents

Generally you will want to write in imperative as this is what automatic commits like merge do.

When should you commit?

Commit early, commit often

- Every time you complete a small change or fix a bug
- You don't normally want to commit broken code (intentionally at least)
- In some instances you might want to auto-commit - but probably not too often.
  - Normally this works if changes can't break something. E.g. Password Manager

Unfortunately, this doesn't have one simple answer.

Some examples of auto-committing are for your password manager.

Commit Messages

In case you hadn't noticed, I quite like Randall Munroe.

I am bad for this, particularly on personal projects.

Commit

```
# Open editor for message
git commit
# Read message from file
git commit -F <file or - for stdin>
# Provide message directly
git commit -m "<message>"
```

```
 git commit
[master (root-commit) 248c2a3] Add greeting.py
 1 file changed, 7 insertions(+)
 create mode 100755 greeting.py
```

Running git commit will open your editor.

I only really use -F if I am doing so from a script

Diff

```
# Diff between last commit and current state
git diff
# Diff between 2 commits or references
git diff commit1..commit2
# Same as above but on a single file
git diff a/file
```

Diff is pretty smart. It will normally work for whatever combinations of commits, references (more on that later) or files.

Diff

```
 git diff
diff --git a/greeting.py b/greeting.py
index 451f386..e73cd5b 100755
--- a/greeting.py
+++ b/greeting.py
@@ -1,7 +1,7 @@
 #!/usr/bin/env python

 def main():
-    print("Hello")
+    print("Hello World")

 if __name__ == "__main__":
     main()
```

These are hopefully quite easy to understand. Red lines mean a line was removed, green means a line was added.

Log

```
❯ git commit -m "Change \"Hello\" to \"Hello World\""
[master 51c696f] Change "Hello" to "Hello World"
 1 file changed, 1 insertion(+), 1 deletion(-)
...
❯ git log
commit 51c696f9c3b7859b290d7d17a4420a4dc096b403
Author: Jonathan Hodgson <git@jonathanh.co.uk>
Date: Mon Jun 15 20:08:02 2020 +0100

    Change "Hello" to "Hello World"

commit 248c2a388534aff85d6155d3bfdae4d3ecc0e67d
Author: Jonathan Hodgson <git@jonathanh.co.uk>
Date: Mon Jun 15 20:08:00 2020 +0100

    Add greeting.py

    Adds the first file, currently always prints Hello
```

Here we see a commit done with the -m flag. I generally only use -m if it is a trivial change like this and there is no need to have a body.

You can see that the log shows a list of the two commits we have made on this project.

The git log command has a lot of flags. We will see some of them later.

Under the hood

```
❯ zlib-flate -uncompress < .git/objects/51/c696f9c3b7859b290d7d17a4420a4dc096b403
commit 256tree a3a29fe10acf63f53164292740d22d530750d9ab
parent 248c2a388534aff85d6155d3bfdae4d3ecc0e67d
author Jonathan Hodgson <git@jonathanh.co.uk> 1592248082 +0100
committer Jonathan Hodgson <git@jonathanh.co.uk> 1592248082 +0100

Change "Hello" to "Hello World"
❯ zlib-flate -uncompress < .git/objects/51/c696f9c3b7859b290d7d17a4420a4dc096b403 | sha1sum
51c696f9c3b7859b290d7d17a4420a4dc096b403
❯ git cat-file -p a3a29fe
100755 blob e73cd5b9fd440608e22b70411a55645e3611fa15 greeting.py
❯ git cat-file -p e73cd5b
#!/usr/bin/env python

def main():
    print("Hello World")

if __name__ == "__main__":
    main()
```

I said earlier that we would be looking at how all this works at quite a low level. This is where that starts.

We can also use the cat-file command built into git to do the same thing. We can see that Commits, trees and blobs are all stored in the same way.

You will also see that you can often use a prefix of the first 4 or more characters of a hash. It is quite common to use the first 7 or 8.

Hopefully you will see from this that the inner workings of git isn't that complicated.

.gitignore

This file tells git which files not to track.

```
*.log
*.doc
*.pem
*.docx
*.jpg
*.jpeg
*.pdf
*.png
.DS_Store/
*.min.css
*.min.js
dist/
```

This will not stop git tracking a file if it's already being tracked.

If you start tracking large binary files, git isn't going to be able to compress them. This will result in a massive repo and a headache for everyone. If at all possible, don't track large files, especially if they are going to be changed. Remember, git stores each version of each file. With text, this is fine as it can be compressed efficiently. If it's not text, it can't.

You should probably also try to avoid including minified files as git won't be able to merge them automatically.

References

- We have just seen that commits are simply (compressed) text files, addressed by a hash.
- References are a way of addressing them without remembering the hash.
- Unlike the hashes, references can change - and they do change.

We've seen a couple of these (sort of)

Master and Head

References

```
git log
commit 51c696f9c3b7859b290d7d17a4420a4dc096b403 (HEAD -> master)
Author: Jonathan Hodgson <git@jonathanh.co.uk>
Date: Mon Jun 15 20:08:02 2020 +0100

    Change "Hello" to "Hello World"

commit 248c2a388534aff85d6155d3bfdae4d3ecc0e67d
Author: Jonathan Hodgson <git@jonathanh.co.uk>
Date: Mon Jun 15 20:08:00 2020 +0100

    Add greeting.py

    Adds the first file, currently always prints Hello
```

There are two references we can see here, master and HEAD.

References

```
❱ cat .git/refs/heads/master
51c696f9c3b7859b290d7d17a4420a4dc096b403
❱ cat .git/HEAD
ref: refs/heads/master
```

- References are stored in the `.git/refs` folder
- The `heads` folder contains references to the heads (or tips) of all local branches

References

HEAD

- The HEAD references is directly in the `.git` folder.
- It refers to the “current” commit. It is how git knows where you are.
- This normally refers to a branch’s head commit.
- In some situations it will refer to a commit directly.

Not sure why it is not in refs folder

If it refers directly to a commit, the repository is in what is called a “detached head” state.

Branches

- Allows multiple features to be developed in parallel without interference.
- Allows multiple people to collaborate easily.

```
# List Branches
git branch # -v adds more info
# Create a branch called test
git branch test # or
cp ~/.git/refs/heads/master ~/.git/refs/heads/test
# Switch to new branch
git switch test # or
git checkout test
# Create and switch in one go
git switch -c test # or
git checkout -b test
```

Branches are represented in git as references in the heads folder.

They can be created by simply creating a file there.

The git checkout command does A LOT of stuff. It can be confusing so it's functionality has been split up into several smaller commands. If you have git 2.23.0 or newer, you will be able to use it.

Be aware that a lot of tutorials etc. will use the checkout command. Version 2.23.0 was released in August 2019.

Branches

```
 git branch -v
* master 51c696f Change "Hello" to "Hello World"
 git switch -c test
 git branch -v
  master 51c696f Change "Hello" to "Hello World"
* test   51c696f Change "Hello" to "Hello World"
 git log --oneline --all
51c696f (HEAD -> test, master) Change "Hello" to "Hello World"
248c2a3 Add greeting.py
```

As we saw, there are numerous ways to create the commit.

What is interesting to note here is that both are still currently pointing at the same commit.

Head is pointing at test so any new commits will be on this branch.

Also take note of the git log command. `--on-line` shows a short version and `--all` shows all branches.

Differing Branches

```
 git switch master
 vim greeting.py
  # CAPITALISE HELLO WORLD #
 git commit -am "Capitalises Hello World"
[master b2e27a0] Capitalises Hello World
 1 file changed, 1 insertion(+), 1 deletion(-)
 git switch test
 vim greeting.py
  # Adds the line "import sys" #
 git commit -am "Adds sys import for arg parsing"
[test 1f16b2d] Adds sys import for arg parsing
 1 file changed, 2 insertions(+)
```

Differing Branches

```
❯ git log --oneline --all --graph
* b2e27a0 (master) Capitalises Hello World
| * 1f16b2d (HEAD -> test) Adds sys import for arg parsing
|/
* 51c696f Change "Hello" to "Hello World"
* 248c2a3 Add greeting.py
❯ git diff mater..test
diff --git a/greeting.py b/greeting.py
index 483ed66..a0ab589 100755
--- a/greeting.py
+++ b/greeting.py
@@ -1,7 +1,9 @@
 #!/usr/bin/env python

+import sys
+
 def main():
- print("HELLO WORLD")
+    print("Hello World")

 if __name__ == "__main__":
     main()
```

This shows what would be needed to take you from master to test.

Notice the `--graph` flag which adds the drawing to the left

Simple Merge

```
❯ git switch master
❯ git merge test
Auto-merging greeting.py
Merge made by the 'recursive' strategy.
 greeting.py | 2 ++
 1 file changed, 2 insertions(+)
❯ git log --oneline --all --graph
*   ab1243e (HEAD -> master) Merge branch 'test'
|\
| * 1f16b2d (test) Adds sys import for arg parsing
* | b2e27a0 Capitalises Hello World
|/
* 51c696f Change "Hello" to "Hello World"
* 248c2a3 Add greeting.py
```

After working on septate branches, you will probably want to merge them eventually.

In this situation, Git was able to work everything out itself.

Tidy Up

```
git switch master
git branch -d test
Deleted branch test (was 1f16b2d).
```

Now that we have finished with that branch, we can delete it.

More Complex merge

```
  # Make changes to 2 branches in the same place #
❯ git switch master
❯ git log --oneline --all --graph
* 1bfb5eb (dog) Makes a dog say Woof
| * 13cc567 (HEAD -> master) Makes a cat say Meow
|/
*   ab1243e Merge branch 'test'
|\
| * 1f16b2d Adds sys import for arg parsing
* | b2e27a0 Capitalises Hello World
|/
* 51c696f Change "Hello" to "Hello World"
* 248c2a3 Add greeting.py
❯ git merge dog
Auto-merging greeting.py
CONFLICT (content): Merge conflict in greeting.py
Automatic merge failed; fix conflicts and then commit the result.
```

At times, git won't be able to merge automatically.

Dealing with merges is something that there are around a million different tools you can use but I think they over complicate what is actually quite a simple process.

More Complex merge

```
cat greeting.py
#!/usr/bin/env python

import sys

<<<<<<< HEAD
def cat():
    print("Meow")

def main():
    if len(sys.argv) > 1 and sys.argv[1] == "cat":
        cat()
=======
def dog():
    print("Woof")

def main():
    if len(sys.argv) > 1 and sys.argv[1] == "dog":
        dog()
>>>>>>> dog
    else:
        print("HELLO WORLD")

if __name__ == "__main__":
    main()
```

Here you can see that the bit(s) git couldn't work out are delimited by `<<<<<<<` and `>>>>>>>` and separated by `=======`.

All you (the programmer) needs to do is fix it.

More Complex merge

```
vim greeting.py
# Fix the conflict(s) #
git add greeting.py
git commit
[master 7b63f4c] Makes a dog say Woof
git log --oneline --all --graph
*   7b63f4c (HEAD -> master) Makes a dog say Woof
|\
| * 1bfb5eb (dog) Makes a dog say Woof
* | 13cc567 Makes a cat say Meow
|/
*   ab1243e Merge branch 'test'
|\
| * 1f16b2d Adds sys import for arg parsing
* | b2e27a0 Capitalises Hello World
|/
* 51c696f Change "Hello" to "Hello World"
* 248c2a3 Add greeting.py
```

Time Travel

```
# Print a version of a file
git show <commit or reference>:<file>
# Restore a file from a previous version
git restore -s <commit or reference> file # or
git checkout <commit or reference> -- file
# Go back in time to a commit
git switch --detach <commit or reference> # or
git checkout <commit or reference>
```

Note that for the show command, you need to provide the full path (relative to the root of the project), not the relative path

Both the restore command and the switch command can also be achieved with checkout on older versions of git.

In the case of restore and checkout, they are not identical. Checkout will also stage the file whereas restore wont (without an additional flag)

Remotes

- The majority of Git commands only affect your local repository.
- Git has a concept called remotes which you can think of as other instances of the same repository
- Git has a selection of commands that are used to communicate with these remote repositories
- It can communicate on multiple protocols including
  - HTTP(S)
  - SSH
  - GIT
  - Local Filesystem

The most common public remote is Git hub. This is one of many options.

The most common protocols are HTTP and SSH, both offering advantages over the other.

HTTP is easier to use and allows for anonymous access (useful for public repositories)

SSH allows for easier authentication but there is no anonymous authentication.

For ease of automating screen shots, I will be using local file system although all commands are the same.

Adding a remote

```
❯ git remote add origin /tmp/demo-remote
❯ git remote
origin
❯ git remote get-url origin
/tmp/demo-remote
```

In this example, I am using a local folder as a URL but you can use anything.

Pushing your code

Long Way

```
git push <remote> <local-branch>:<remote-branch>
# E.g.
git push origin master:master
```

Although this is usable, it is a bit annoying since you will normally want to create a 1 to 1 correspondence between your local branches and the remote branches.

Pushing your code

Easy way

```
❯ git branch --set-upstream-to=origin/master master
Branch 'master' set up to track remote branch 'master' from 'origin'.
❯ git push
❯ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

Admittedly, this looks like a longer way. However, the first command only needs to be run the once. Git will then remember that the remote branch origin/master should be linked with the local branch master.

As you can see from the git-status now, it tells us that the current branch is up to date with the remote branch.

Retrieving changes from the remote

```
❯ git fetch
❯ git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
❯ git log --oneline --all --graph
* 959f606 (origin/master) Adds Cow option
*   7b63f4c (HEAD -> master) Makes a dog say Woof
|\
| * 1bfb5eb (dog) Makes a dog say Woof
* | 13cc567 Makes a cat say Meow
|/
*   ab1243e Merge branch 'test'
|\
| * 1f16b2d Adds sys import for arg parsing
* | b2e27a0 Capitalises Hello World
|/
* 51c696f Change "Hello" to "Hello World"
* 248c2a3 Add greeting.py
❯ git merge
Updating 7b63f4c..959f606
Fast-forward
 greeting.py | 5 +++++
 1 file changed, 5 insertions(+)
```

Here another user has pushed changes to the master branch.

You can see here that the git fetch command downloads the required information. However, it doesn't change your working tree.

In order to update our local master, we can simply do a git merge

Also note that after the git fetch command, the git status gives us some useful information. This is another advantage of setting the corresponding upstream branch.

- We are 1 commit behind the remote version
- The branch can be fast forwarded.

Git Pull

Shortcut

```
git pull
git pull <remote> <branch>
```

Git has a command called pull which does the equivalent of a fetch then a merge.

The short version will only work if you have the local branch linked with a remote branch.

Cloning

```
# Clone a repository into a folder
git clone <URL> <folder>

# Clone a repository into a folder on a specific branch
git clone --branch <branch> <URL> <folder>

# Shallow clone a repository into a folder
git clone --shallow <URL> <folder>
```

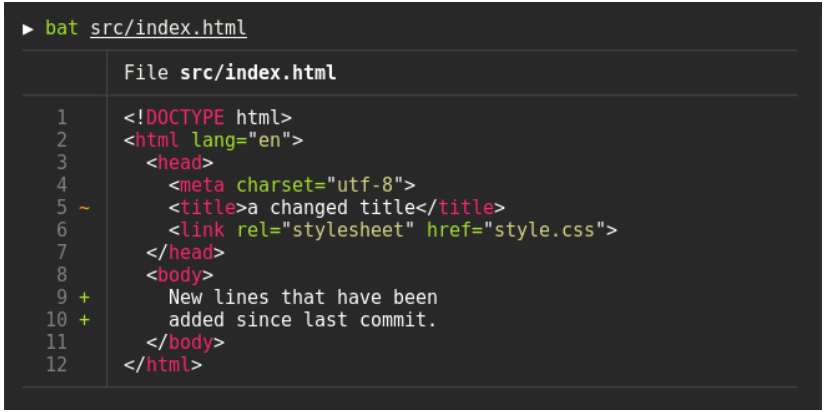
If you would like to start from an existing remote repository, you can use the git clone command.

This will automatically set up the link between the local and remote branches.

A shallow clone doesn't download all of the history.

Useful supporting tools

Bat



https://github.com/sharkdp/bat

Bat is described as cat with wings.

It adds syntax highlighting to files. Useful even if you're not using Git

As this is a git talk, it shows lines that have changed since the last commit

Useful supporting tools

RigGrep / Fd

https://github.com/sharkdp/fd

https://github.com/BurntSushi/ripgrep

Alternatives to grep and find

Fd, in particular, is not a full replacement for find but does most of what you want

Both (by default) will respect your gitignore file.

Useful supporting tools

Delta

jedi/evaluate/names.py

```
class AbstractNameDefinition(object):
28
        return {self}

    @abstractmethod
    def get_qualified_names(self):
    def get_qualified_names(self, include_module_names=False):
        raise NotImplementedError

    def get_root_context(self):
class AbstractTreeName(AbstractNameDefinition):
53
        self.parent_context = parent_context
        self.tree_name = tree_name

    def get_qualified_names(self):
    def get_qualified_names(self, include_module_names=False):
        import_node = search_ancestor(self.tree_name, 'import_name', 'import_from')
        if import_node is not None:
            return tuple(n.value for n in import_node.get_path_for_name(self.tree_name))

        parent_names = self.parent_context.get_qualified_names()
        if parent_names is None:
            return None
        return parent_names + [self.tree_name.value]
        parent_names += (self.tree_name.value,)
        if include_module_names:
            module_names = self.get_root_context().string_names
            if module_names is None:
                return None
            return module_names + parent_names
        return parent_names

    def goto(self):
        return self.parent_context.evaluator.goto(self.parent_context, self.tree_name)
```

https://github.com/dandavison/delta

This is a tool that can make your diff output look better.

Useful supporting tools

BFG Repo Cleaner

You'll need something like this when you realise you have just committed your ssh keys

https://rtyley.github.io/bfg-repo-cleaner/

For the time that you accidentally commit your ssh keys.

I accidentally committed a database for an Woocommerce site.

Useful supporting tools

Shell Integration

Git ships with completion for bash, zsh and tcsh. You may need to source it in the relevant rc file.

Prompt customisation is available out of the box for bash and zsh.

If you haven't ever tried zsh, give it a shot. Tab completion is so much more useful than Bash's.

Useful supporting tools

Pass

- Password Manager
- Uses Git for keeping track of history
- Syncs using Git
- Everything is encrypted with a GPG key
- Has compatible android, ios and browser apps.

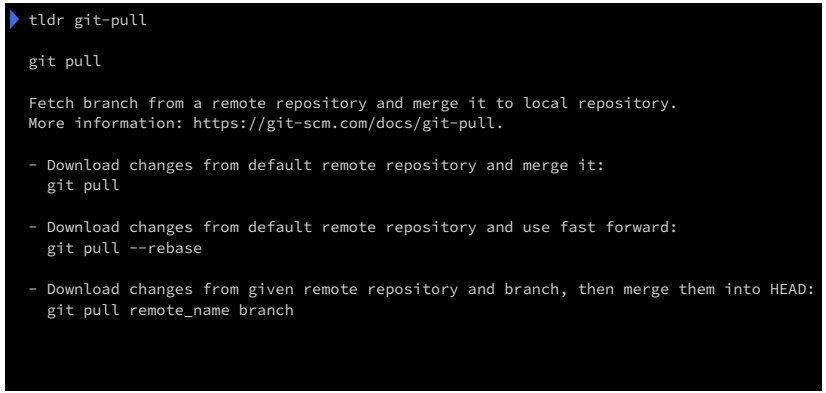
https://www.passwordstore.org/

A password manager that uses Git for sync and history.

Useful supporting tools

tldr

The man page for git pull is over 700 lines.



https://github.com/tldr-pages/tldr

Sometimes incredibly thorough documentation isn't what you want. Sometimes you know the command you need but you can't remember exactly how to use it.

This is not specific to git commands, it covers a LOT.