Git: Version control

How do we track how code changes?

Jo Hallett

October 13, 2025

Once upon a time…

In the distant past I wrote some code.

The next day Jo wrote more code...

The next day I wrote some more code.

- But being a forgetful sort I forgot what I had written.
- Never mind keep hacking!

And on the third day…

I saw that the code was good and released it as version 1.0.

Matt lends a hand!

Matt says my code is pretty neat!
He wants to add a cool new feature to my code!

- Thanks Matt!

On the fifth day… Jo went back to work

Gotta start working towards Jo's code 2.0!

Matt finds a bug!

On the sixth day Matt finds a bug in the 1.0 release.

- Matt doesn't want to wait for the 2.0 release and needs a fix now!
- Better go back to the old code and fix it there!

So now I have:

- Version 1.0
- Version 1.0 + Backported fix for Matt
- Code I'm working onn

And on the seventh day…

I have to go have a lie down.

- Tracking versions of things is hard!

And from then on it only got worse…

- And then Matt wanted to relicense his code so I needed to figure out which lines he wrote.
- And then I lost my laptop and needed a backup.
- And then I was in the Swiss mountains and needed to work remotely on a copy.
- And then I left my laptop in a chalet and had to wait for it to be sent back to me.
- Then when I got my laptop back I had to somehow combine the work on the laptop with other things I'd been working on.
- And then I needed to figure out when I'd introduced a bug to see which releases I needed to fix.

I copied this slide from last year…

I am **incredibly** lazy.

- Managing things by hand seems like an absolute pain.
- If I can get a computer to do it for me, do that.
- I'll even go out of my way to learn tools that'll let me be lazier later.

Version control

Version control tracks changes in source code

- Or any file really (it just works especially well for plain text)

Instead of manually copying code and renaming things `version-1.2+matt-backport`

- Version control tracks changes
- Lets others interact with it by distributing code

(It can do other things too but this is the core).

The one we use is by this guy…

Git

Back in 2005, Linus Torvalds invented Git to manage the Linux kernel source.

- They were using something called BitKeeper.
- They wanted to charge money for it.
- Linus rewrote his own tool more suited to the needs of the kernel.

It wasn't the first version control system.

- RCS back in 1982(?)
- There are more modern version control systems that do things better.
- It is the version control system we've standardized on.
- You still sometimes have to use the other ones.

(Mercurial/Fossil/Subversion/CVS/Darcs)

Once upon a time... (with Git!)

In the distant past I wrote some code.

```
git add mycode.c
git commit -m "Initial␣commit␣of␣my␣cool␣code"
```

The next day Jo wrote more code... (with Git!)

The next day I wrote some more code.

```
git add mycode.c && git commit -m "More work done"
```

Being a sort I forgot what I had written.

- ~~Never mind keep hacking!~~

```
git log mycode.c
```

```
commit e9e1e48b8250d03aaab6c1af41195743f7adcedf
Author: Jo Hallett <joseph.hallett@bristol.ac.uk>
Date:  Wed Feb 7 15:11:33 2024 +0000

    Adds distributed authentication using Blockchain and GPT-4!
  git revert e9e1e48b
```

And on the third day… (with Git!)

I saw that the code was good and released it as version 1.0.

```
git tag version-1.0
```

Matt lends a hand! (with Git!)

Matt says my code is pretty neat!
He wants to add a cool new feature to my code!

- Thanks Matt!

```
git apply cool-feature-from-matt.patch
```

On the fifth day… Jo went back to work (with Git!)

Gotta start working towards Jo's code 2.0!

```
git add mycode.c mylibrary.c
git commit -m "Refactored␣my␣code␣to␣add␣a␣library!"
```

Matt finds a bug! (with Git!)

On the sixth day Matt finds a bug in the 1.0 release.

- Matt doesn't want to wait for the 2.0 release and needs a fix now!
- Better go back to the old code and fix it there!

```
git checkout version-1.0 # Go back to 1.0
git branch hotfix-for-matt # Create a branch of work based on version-1.0
git checkout hotfix-for-matt # Switch to it
git add mycode.c # Add the changes needed
git commit "Fixes␣Matt's␣bug" # ...and commit to them
git format-patch version-1.0 # Create a patch of all the changes
git checkout main # Go back to working on the new one
```

And on the seventh day… (with Git!)

I have to go have a lie down.

- Honestly, who works on weekends?

And from then on it only got ~~worse~~ better… (with Git!)

- And then Matt wanted to relicense his code so I needed to figure out which lines he wrote.

```
git blame
```

- And then I lost my laptop and needed a backup.

```
git push
```

- And then I was in the Swiss mountains and needed to work remotely on a copy.

```
git pull
```

- And then I left my laptop in a chalet and had to wait for it to be sent back to me.
- Then when I got my laptop back I had to somehow combine the work on the laptop with other things I'd been working on.

```
git merge
```

- And then I needed to figure out when I'd introduced a bug to see which releases I needed to fix.

```
git bisect
```

Is there a manual?

```
apropos git # or man -k git
```

- git(1) - the stupid content tracker
- git-add(1) - Add file contents to the index
- git-commit(1) - Record changes to the repository
- git-help(1) - Display help information about Git
- gitcli(7) - Git command-line interface and conventions
- gitcore-tutorial(7) - A Git core tutorial for developers
- gitcredentials(7) - Providing usernames and passwords to Git
- giteveryday(7) - A useful minimum set of commands for Everyday Git
- gitfaq(7) - Frequently asked questions about using Git
- gitglossary(7) - A Git Glossary
- gitsubmodules(7) - Mounting one repository inside another
- gittutorial(7) - A tutorial introduction to Git
- gittutorial-2(7) - A tutorial introduction to Git: part two
- gitworkflows(7) - An overview of recommended workflows with Git

(and about 200 other manual pages…)

Welcome to Git

Git is the defacto standard tool for managing changes to code.

- It has a reputation for being confusing to beginners.
- Once you get used to it, it's kinda fine.

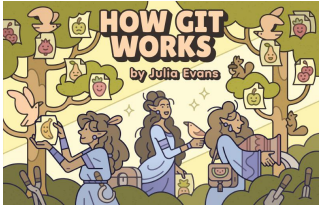
There are GUIs to make things easier…

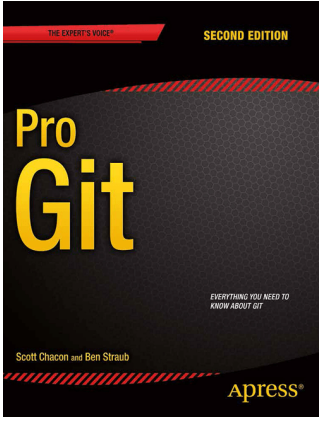
- We're **not** gonna teach them.
- Sooner or later you need to know how to do something moderately complex and you'll have to use a commandline.
- (I use magit, but I like Emacs…)

What are we gonna do today?

I'm gonna show you the absolute basics for using Git

- You have to go away and practice (coming to labs is a good start ;-))
- The `man` pages are really good (get used to using them!)





Lets create a git repo!

```
$ mkdir /tmp/tutorial
$ cd /tmp/tutorial
$ git init
```

```
Initialized empty Git repository in /tmp/tutorial/.git/
```

```
$ ls -lA
```

```
total 4
drwxr-xr-x 7 joseph wheel 512 Jul 1 15:24 .git
```

```
$ git status
```

```
On branch main

No commits yet

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

Lets create a file!

```
$ ed hello.c
i
#include <stdio.h>
int main(void) {
  print("Hello␣World!\n");
  return 0;
}
.
w
q
```

```
0
77
```

(Note: Please don't actually use `ed` unless you want to scare CS students ;-))

Then status...

```
$ git status
```

```
On branch main

No commits yet

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

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

Lets add something!

```
$ git add hello.c
```

```
$ git status
```

```
On branch main

No commits yet

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

At this stage Git is saying it knows about `hello.c` at the point when you added it… and that it is a `new file`.
…but it isn't committed yet!

Lets commit something!

```
$ git commit -m "Added␣a␣hello␣world␣program!"
```

```
[main (root-commit) 6c93433] Added a hello world program!
 1 file changed, 5 insertions(+)
 create mode 100644 hello.c
```

```
$ git status
```

```
On branch main
nothing to commit, working tree clean
```

Did we do anything?

```
$ git log
```

```
commit 6c93433eb00459586244c05f255da3497cf37ac6
Author: Jo Hallett <joseph.hallett@bristol.ac.uk>
Date:  Mon Jul 1 15:24:34 2024 +0100

    Added a hello world program!
```

```
$ git status
```

```
On branch main
nothing to commit, working tree clean
```

Lets add another file!

```
$ ed Makefile
i
all: hello
.
w
q
```

```
0
11
```

```
$ git status
```

```
On branch main
Untracked files:
  (use "git␣add␣<file>..." to include in what will be committed)
      Makefile

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

And commit

```
$ git add Makefile
$ git commit -m "Adds␣a␣build␣script"
```

```
[main d7605b9] Adds a build script
 1 file changed, 1 insertion(+)
 create mode 100644 Makefile
```

```
$ git log --pretty=reference
```

```
d7605b9 (Adds a build script, 2024-07-01)
6c93433 (Added a hello world program!, 2024-07-01)
```

And if we now try and build it?

```
$ make
```

```
cc -O2 -pipe -o hello hello.c
hello.c:3:3: warning: call to undeclared function 'print'; ISO C99 and later do not support implicit function d
  print("Hello␣World!\n");
  ^
1 warning generated.
ld: error: undefined symbol: print
>>> referenced by hello.c
>>>            /tmp/hello-04b176.o:(main)
>>> did you mean: printf
>>> defined in: /usr/lib/libc.so.100.1
cc: error: linker command failed with exit code 1 (use -v to see invocation)
*** Error 1 in /tmp/tutorial (<sys.mk>:85 'hello')
```

Lets fix it!

```
$ ed hello.c
3s/print/printf/
%n
w
q
```

```
77
1       #include <stdio.h>
2       int main(void) {
3         printf("Hello␣World!\n");
4         return 0;
5       }
78
```

```
$ git status
```

```
On branch 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: hello.c

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

Lets check what we changed

```
$ git diff HEAD
```

```
diff --git a/hello.c b/hello.c
index 7f6a938..f77c0bc 100644
--- a/hello.c
+++ b/hello.c
@@ -1,5 +1,5 @@
 #include <stdio.h>
 int main(void) {
-  print("Hello␣World!\n");
+  printf("Hello␣World!\n");
   return 0;
 }
```

```
$ git add hello.c
$ git commit -m "Fixes␣call␣to␣printf"
```

```
[main b3bedd5] Fixes call to printf
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Git remotes and forges

Git remotes are places where you can send your code to.

- If you configure them right other people can also get your code directly from them
- Just a folder you can send files to really
  - So could be a directory on your computer
  - Somewhere accessible via SSH
  - A webserver that accepts POST requests

A Git forge is a place where you can host a remote

- Often have features to help you collaborate
- Microsoft's `https://github.com` is common
- I like one called `https://sr.ht` (and I pay for it)

You can also create your own!

- See 4.2 Git on the Server in the Pro Git book
- (Basically just funky access control and SSH)

What remotes do we have?

```
$ git remote -v
```

Lets add one!

```
$ git remote add origin git@git.sr.ht:~sherbert/demo-git-repo
$ git remote -v
```

```
origin git@git.sr.ht:~sherbert/demo-git-repo (fetch)
origin git@git.sr.ht:~sherbert/demo-git-repo (push)
```

Woosh!

And lets send our code there!

```
$ git push
```

```
Host key fingerprint is SHA256:WXXNZu0YyoE3KBl5qh4GsnF1vR0NeEPYJAiPME+P09g
+--[ED25519 256]--+
|    o o..*.O=o.o.|
|     =.O* BoO.o *|
|     .*.E+.* * * |
|  o o ..o. + . .|
|   = . .S      |
|  .   +        |
|     o .       |
|      .        |
|               |
+----[SHA256]-----+

remote: Default branch updated to main
To git.sr.ht:~sherbert/demo-git-repo
 * [new branch]   main -> main
```

If someone else wants our code…

They can fetch it directly from the remote if they have a URL to it

```
$ git clone https://git.sr.ht/~sherbert/demo-git-repo /tmp/demo-git-repo
```

```
Cloning into '/tmp/demo-git-repo'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 9 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (9/9), done.
```

```
$ cd /tmp/demo-git-repo
$ git log --pretty=oneline
```

```
b3bedd5a69706b984f7a5e6844dd741fec787b8c (HEAD -> main, origin/main, origin/HEAD) Fixes call to printf
d7605b93ce0f117bdf0183bc3d79c567880238ee Adds a build script
6c93433eb00459586244c05f255da3497cf37ac6 Added a hello world program!
```

What is HEAD?

First line of that log says:

`(HEAD -> main, origin/main, origin/HEAD)`

What is that about, and what are those weird numbers?

The big hexadecimal string is the commit id and refers to a single commit.

- It's a SHA hash of the commit and all its metadata, and the previous commit.
- If you want you can give them more meaningful names with `git tag` (useful for versioning numbers!)

`HEAD` is a pointer to the last commit you checked out:

- Either the last commit
- The place you explicitly checked out with the `git checkout` command
- Work you haven't committed can't be pointed to

`main` (and origin/main) are branches…

Branching

A branch is a series of commits that represent a set of work
The default branch is called main (or sometimes master)

- As you add commits they are added on top of the `HEAD` pointer
- If the `HEAD` is tracking a branch, the branch (and the HEAD pointer) are updated to point to the commit at the top of the branch
- `HEAD -> main` means that the `HEAD` pointer is tracking `main`

You can call branches whatever you like, and you can have as many as you like

- BUT you should try and ensure that whatever is commited on the main branch always works
- While your developing and working on code create a new branch from the last known good point and work there
- (I'll show you how to merge in work next week!)

```
$ git checkout -b development
```

```
Switched to a new branch 'development'
```

Remote branches

`origin/main` is telling you that Git is aware that at the remote `origin` there is also a branch called `main`.

- Last time `git fetch`-ed from that remote, the main branch at origin was pointing there
- And so was the `HEAD` at the `origin`

`main` and `origin/main` are not the same branch

- But they may have a similar history

Things may have changed though over time… to update the remote branches:

```
$ git fetch --all
```

Looks like nothing has changed… til next week!

Recap

Today we talked about:

- Why version control is useful
- How to create a new Git repo (`git init`)
- How to add files (`git add`)
- How to commit files (`git commit`)
- How to check on a Git repo (`git status`)
- How to see the history of a Git repo (`git log`)
- Git remotes (like Github) (`git remote`)
- Sending changes to Git remotes (`git push`)
- How to copy a Git remote (`git clone`)
- Fetching changes from Git remotes (`git fetch`)
- Git branches (`git branch`)
- Where to get help (`apropos git`)

Next time we'll talk about:

- Merging changes
- What happens when it goes wrong
- Sending patches and pull requests to other people
- Weirder Git commands

Practice makes perfect…

- None of this stuff is natural.
- Everyone struggles a bit with it.
- I've been using it for ~20 years and there are things I still get wrong…

If you practice it **does get easier**.

- (Come to labs and we can help you!)

You are going to be using Git every day if you work in software.

- And if you see people sending you copies of files like `Company-report-version-2.3-FINAL-JH-checked.docx`…