

# Getting Git

by Daniel Cox

hello

I'm daniel cox, a software engineer on i2ps, and this is Getting Git.  
I asked to give a talk on git because I believe it to be brilliant but misunderstood  
software with a high-learning curve, and I think every developer would benefit from a  
deeper understanding of it

# Getting Git

by Daniel Cox

(who is enormously  
grateful to Scott Chacon  
for half of these slides)

half my slides come from a presentation of Scott Chacon's,  
which I'll mention again at the end

since the talk ended up being shorter than I expected, please feel free to interrupt  
me with questions

## THE MASTER PLAN

- What is Git?
- Understanding Git
  - The Basics
  - The Git Object Database
  - History Inspection
  - Branching Workflows
  - Collaboration
  - Advanced Stuff
  - Troubleshooting
- Review
- Resources
- Questions

here is the master plan

# THE MASTER PLAN

- What is Git? 3 minutes
- Understanding Git 15 hours
  - The Basics
  - The Git Object Database
  - History Inspection
  - Branching Workflows
  - Collaboration
  - Advanced Stuff
  - Troubleshooting
- Review 2 minutes
- Resources 3 minutes
- Questions 20 minutes

and here is the approximate breakdown.

the talk is mostly meant for those of you who already use git but don't love it yet,  
though I do have a few slides on what git *is*

## **What *is* Git?**

- Git is a Version Control System
  - Versioning
  - Collaboration
- Other things Git is:
  - A very simple content tracker
  - A key/value object database with a VCS front-end
  - A toolkit

git, first, is a version control system. it was designed to solve two major problems  
versioning - you need to keep history of your content so you can e.g., go back... or  
branch off to try new things... remember what you've done... try to figure out what  
someone else has done...

collaboration - you need to work together with other people on a project without  
emailing files back and forth or overwriting each other's changes

other things git is: a very simple content tracker, a key/value object database with a  
vcs front-end, and a toolkit

## *What is Git?*

"I'm an egotistical b[-----], and I name all my projects after myself. First 'Linux', now 'git'."

-- Linus Torvalds



it's also a creation of linus torvalds, to serve the needs of the linux kernel

git-add	git-fast-export	git-merge-recur	git-revert
git-add--interactive	git-fast-import	git-merge-recursive	git-rm
git-am	git-fetch	git-merge-recursive-old	git-runstatus
git-annotate	git-fetch--tool	git-merge-resolve	git-send-email
git-apply	git-fetch-pack	git-merge-stupid	git-send-pack
git-applymbox	git-filter-branch	git-merge-subtree	git-sh-setup
git-applypatch	git-fmt-merge-msg	git-merge-tree	git-shell
git-archimport	git-for-each-ref	git-mergetool	git-shortlog
git-archive	git-format-patch	git-mktag	git-show
git-bisect	git-fscck	git-mktree	git-show-branch
git-blame	git-fscck-objects	git-mv	git-show-index
git-branch	git-gc	git-name-rev	git-show-ref
git-bundle	git-get-tar-commit-id	git-pack-objects	git-ssh-fetch
git-cat-file	git-grep	git-pack-redundant	git-ssh-pull
git-check-attr	git-gui	git-pack-refs	git-ssh-push
git-check-ref-format	git-hash-object	git-parse-remote	git-ssh-upload
git-checkout	git-http-fetch	git-patch-id	git-stash
git-checkout-index	git-http-push	git-peek-remote	git-status
git-cherry	git-imap-send	git-prune	git-stripespace
git-cherry-pick	git-index-pack	git-prune-packed	git-submodule
git-citool	git-init	git-pull	git-svn
git-clean	git-init-db	git-push	git-svnhimport
git-clone	git-instaweb	git-quiltimport	git-symbolic-ref
git-commit	git-local-fetch	git-read-tree	git-tag
git-commit-tree	git-log	git-rebase	git-tar-tree
git-config	git-lost-found	git-rebase--interactive	git-unpack-file
git-convert-objects	git-ls-files	git-receive-pack	git-unpack-objects
git-count-objects	git-ls-remote	git-reflog	git-update-index
git-cvsexportcommit	git-ls-tree	git-relink	git-update-ref
git-cvsimport	git-mailinfo	git-remote	git-update-server-inf
git-cvsserver	git-mailsplit	git-repack	git-upload-archive
git-daemon	git-merge	git-repo-config	git-upload-pack
git-describe	git-merge-base	git-request-pull	git-var
git-diff	git-merge-file	git-rerere	git-verify-pack
git-diff-files	git-merge-index	git-reset	git-verify-tag
git-diff-index	git-merge-octopus	git-resolve	git-web-browse
git-diff-stages	git-merge-one-file	git-rev-list	git-whatchanged
git-diff-tree	git-merge-ours	git-rev-parse	git-write-tree

Git is a toolkit of about 152 commands...

git-fast-export	git-merge-recur
git-fast-import	git-merge-recursive
	git-merge-recursive-old git-runstatus
git-fetch--tool	git-merge-resolve
git-fetch-pack	git-merge-stupid
git-applybox	git-filter-branch
git-applypatch	git-fmt-merge-msg
git-archimport	git-for-each-ref
	git-mergetool
	git-mktag
	git-fsck
	git-fsck-objects
git-bundle	git-get-tar-commit-id
git-cat-file	
git-check-attr	
git-check-ref-format	git-hash-object
	git-http-fetch
git-checkout-index	git-http-push
git-cherry	git-imap-send
	git-index-pack
	git-init-db
	git-local-fetch
git-commit-tree	
git-config	git-lost-found
git-convert-objects	git-ls-files
git-count-objects	git-ls-remote
git-cvsexportcommit	git-ls-tree
git-cvsimport	git-mailinfo
git-cvsserver	git-mailsplit
git-describe	git-merge-base
	git-merge-file
git-diff-files	git-merge-index
git-diff-index	git-merge-octopus
git-diff-stages	git-merge-one-file
git-diff-tree	git-merge-ours
	git-receive-pack
	git-reflog
	git-relink
	git-request-pull
	git-rerere
	git-resolve
	git-rev-list
	git-rev-parse
	git-repack
	git-repo-config
	git-unpack-file
	git-update-index
	git-update-ref
	git-update-server-inf
	git-upload-archive
	git-upload-pack
	git-var
	git-verify-pack
	git-verify-tag
	git-web-browse
	git-whatchanged
	git-write-tree

most of which are not useful to humans, and exist mostly to be run by other commands and scripts.

git-fast-export	git-merge-recur
git-fast-import	git-merge-recursive
git-fetch--tool	git-merge-recursive-old
git-fetch-pack	git-runstatus
git-applybox	git-merge-resolve
git-applypatch	git-merge-stupid
git-archimport	git-merge-subtree
	git-merge-tree
git-fck	git-mergetool
git-fsck-objects	git-mktag
git-bundle	git-mktree
git-cat-file	git-name-rev
git-check-attr	git-pack-objects
git-check-ref-fc	git-pack-redundant
git-checkout-index	git-pack-rs
git-cherry	git-prune
	git-prune-packed
git-commit-tree	git-read-tree
git-config	git-receive-pack
git-convert-objects	git-quiltimport
git-count-objects	git-read-tree
git-cvsexportcommit	git-relink
git-cvsimport	git-remote
git-cvsserver	git-repack
git-describe	git-repo-config
git-diff-files	git-request-pull
git-diff-index	git-rerere
git-diff-stages	git-resolve
git-diff-tree	git-rev-list
	git-rev-parse
git-fast-export	git-svn
git-fast-import	git-svimport
git-fetch--tool	git-symbolic-ref
git-fetch-pack	git-tar-tree
git-applybox	git-unpack-file
git-applypatch	git-unpack-objects
git-archimport	git-update-index
	git-update-ref
git-fck	git-update-server-inf
git-fsck-objects	git-upload-archive
git-bundle	git-upload-pack
git-cat-file	git-var
git-check-attr	git-verify-pack
git-check-ref-fc	git-verify-tag
git-checkout-index	git-web-browse
git-cherry	git-whatchanged
	git-write-tree

# the “plumbing”

These are referred to as “the plumbing commands”, and we’ll only get into a couple of these

git-add	git-fast-export	git-merge-recur	git-revert
git-add--interactive	git-fast-import	git-merge-recursive	git-rm
git-am	git-fetch	git-merge-recursive-old	git-runstatus
git-annotate	git-fetch--tool	git-merge-resolve	git-send-email
git-apply	git-fetch-pack	git-merge-stupid	git-send-pack
git-applymbox	git-filter-branch	git-merge-subtree	git-sh-setup
git-applypatch	git-fmt-merge-msg	git-merge-tree	git-shell
git-archimport	git-for-each-ref	git-mergetool	git-shortlog
git-archive	git-format-patch	git-mktag	git-show
git-bisect	git-fscck	git-mktree	git-show-branch
git-blame	git-fscck-objects	git-mv	git-show-index
git-branch	git-gc	git-name-rev	git-show-ref
git-bundle	git-get-tar-commit-id	git-pack-objects	git-ssh-fetch
git-cat-file	git-grep	git-pack-redundant	git-ssh-pull
git-check-attr	git-gui	git-pack-refs	git-ssh-push
git-check-ref-format	git-hash-object	git-parse-remote	git-ssh-upload
git-checkout	git-http-fetch	git-patch-id	git-stash
git-checkout-index	git-http-push	git-peek-remote	git-status
git-cherry	git-imap-send	git-prune	git-stripespace
git-cherry-pick	git-index-pack	git-prune-packed	git submodule
git-citool	git-init	git-pull	git-svn
git-clean	git-init-db	git-push	git-svnhimport
git-clone	git-instaweb	git-quiltimport	git-symbolic-ref
git-commit	git-local-fetch	git-read-tree	git-tag
git-commit-tree	git-log	git-rebase	git-tar-tree
git-config	git-lost-found	git-rebase--interactive	git-unpack-file
git-convert-objects	git-ls-files	git-receive-pack	git-unpack-objects
git-count-objects	git-ls-remote	git-reflog	git-update-index
git-cvsexportcommit	git-ls-tree	git-relink	git-update-ref
git-cvsimport	git-mailinfo	git-remote	git-update-server-inf
git-cvsserver	git-mailsplit	git-repack	git-upload-archive
git-daemon	git-merge	git-repo-config	git-upload-pack
git-describe	git-merge-base	git-request-pull	git-var
git-diff	git-merge-file	git-rerere	git-verify-pack
git-diff-files	git-merge-index	git-reset	git-verify-tag
git-diff-index	git-merge-octopus	git-resolve	git-web-browse
git-diff-stages	git-merge-one-file	git-rev-list	git-whatchanged
git-diff-tree	git-merge-ours	git-rev-parse	git-write-tree

```

git-add          git-revert
git-add--interactive git-rm
git-am           git-send-email
git-annotate
git-apply

git-archive      git-format-patch
git-bisect
git-blame
git-branch       git-gc
git-grep
git-gui

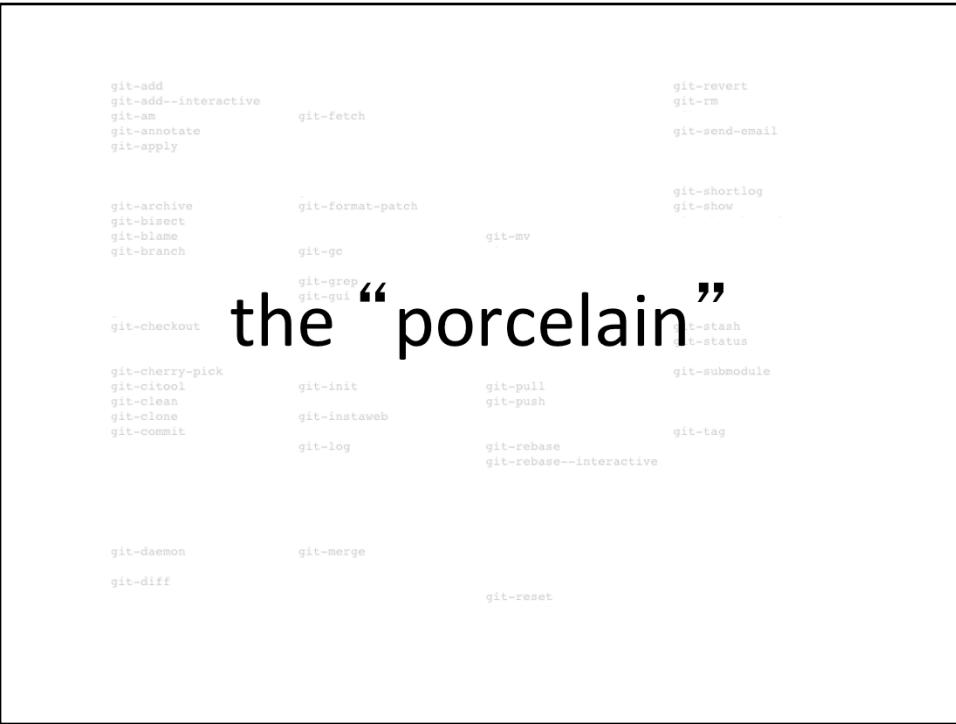
git-checkout
git-cherry-pick
git-citool
git-clean
git-clone
git-commit       git-init
git-instaweb
git-log          git-pull
git-push
git-shortlog
git-show
git-mv

git-stash
git-status
git-submodule
git-tag
git-rebase
git-rebase--interactive

git-daemon
git-diff         git-merge
git-reset

```

However, there are about 40 commands which human users are expected to need frequently at the command line, to perform most of the useful features git provides for your daily work.



These are “the porcelain commands”.

```
git-add          git-revert
git-add--interactive git-fetch      git-rm
git-am           git-format-patch git-send-email
git-annotate     git-grep        git-shortlog
git-apply         git-gui         git-show
                  git-archive    git-mv
                  git-bisect
                  git-blame       git-gc
                  git-branch
                  git-grep
                  git-gui
git-checkout     git-init        git-stash
                  git-cherry-pick git-push      git-status
                  git-citool
                  git-clean
                  git-clone
                  git-commit
                  git-log         git-submodule
                  git-instaweb
                  git-rebase
                  git-rebase--interactive
                  git-tag
                  git-reset
git-daemon
git-diff
git-merge
```

We'll cover about half of these, and touch briefly on more.

## What *is* Git?

Something Git is *not*: Subversion

- Much faster for a slightly larger disk footprint  
(stores snapshots instead of deltas)
- Can do a bunch of stuff without a network connection
- Cheap branching and merging
- No .svn directory clutter
- And quite a bit more, since Git was *not* designed to be the latest evolution of rcs -> cvs -> svn -> ...

here's something git is *not* - it's not subversion

git is much faster, you can work on an airplane, branching isn't something you need to have a meeting about, git drops in *one* hidden directory  
and much much more!

Git uses a lot of the same vocabulary as Subversion, but a different dictionary. The words have quite different meanings.

(checkout, commit, branch...etc. - they mean completely different things in git than they do in subversion)

## *What is Git?*

What Git is on your hard drive.

```
$> mkdir foobar  
$> cd foobar  
$> git init  
  
$> tree -a .git
```

Git hides out of your way until you need it, in a hidden directory in the root of your project folder: `.git`

here we make an empty foobar directory, go into it, and run git init to make it a repository. now we'll look inside the new `.git` directory

## What *is* Git?

```
$> tree -a .git
.git
├── HEAD
├── config
├── description
├── hooks
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── pre-rebase.sample
│   └── prepare-commit-msg.sample
        └── update.sample
└── info
    └── exclude
└── objects
    ├── info
    └── pack
└── refs
    ├── heads
    └── tags

8 directories, 12 files
```

a fresh git repository is tiny. more than half of the clutter here is inoperative sample files

## What is Git?

```
$> touch README
$> git add .
$> git commit -m
"initial commit"
$> tree -a .git
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
│   ├── ...
│   └── ...
├── index
├── info
│   └── exclude
├── logs
│   ├── ...
├── objects
│   ├── 54
│   │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
│   ├── cc
│   │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
│   ├── e6
│   │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
│   ├── info
│   └── pack
└── refs
    ├── heads
    │   └── master
    └── tags

14 directories, 20 files
```

I'd also like you to see that as your repository grows, it gets larger but not much more complicated. When you add a file and commit, the resulting repo looks like this. As we'll talk about soon when we talk about the git object database,

## What is Git?

```
$> touch README  
$> git add .  
$> git commit -m  
"initial commit"  
  
$> tree -a .git  
.  
├── COMMIT_EDITMSG  
├── HEAD  
├── config  
├── description  
├── hooks  
│   ├── ...  
├── index  
├── info  
│   └── exclude  
├── logs  
│   ├── ...  
├── objects  
│   ├── 54  
│   │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd  
│   ├── cc  
│   │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52  
│   ├── e6  
│   │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391  
│   ├── info  
│   └── pack  
└── refs  
    ├── heads  
    │   └── master  
    └── tags  
  
14 directories, 20 files
```

this (54) contains the directory information,

## What is Git?

```
$> touch README  
$> git add .  
$> git commit -m  
"initial commit"  
  
$> tree -a .git  
.  
+-- .git  
    +-- COMMIT_EDITMSG  
    +-- HEAD  
    +-- config  
    +-- description  
    +-- hooks  
    |    +-- ...  
    +-- index  
    +-- info  
    |    +-- exclude  
    +-- logs  
    |    +-- ...  
    +-- objects  
    |    +-- 54  
    |    |    +-- 3b9bebdc6bd5c4b22136034a95dd097a57d3dd  
    |    +-- cc  
    |    |    +-- 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52  
    |    +-- e6  
    |    |    +-- 9de29bb2d1d6434b8b29ae775ad8c2e48c5391  
    +-- info  
    +-- pack  
    +-- refs  
        +-- heads  
        |    +-- master  
        +-- tags  
  
14 directories, 20 files
```

this (cc) contains the commit information,

## What is Git?

```
$> touch README  
$> git add .  
$> git commit -m  
"initial commit"  
  
$> tree -a .git  
.  
+-- .git  
    +-- COMMIT_EDITMSG  
    +-- HEAD  
    +-- config  
    +-- description  
    +-- hooks  
    |    +-- ...  
    +-- index  
    +-- info  
    |    +-- exclude  
    +-- logs  
    |    +-- ...  
    +-- objects  
    |    +-- 54  
    |    |    +-- 3b9bebdc6bd5c4b22136034a95dd097a57d3dd  
    |    +-- cc  
    |    +-- 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52  
    |    +-- e6  
    |    |    +-- 9de29bb2d1d6434b8b29ae775ad8c2e48c5391  
    |    +-- info  
    +-- pack  
    +-- refs  
        +-- heads  
        |    +-- master  
        +-- tags  
  
14 directories, 20 files
```

and this (e6) is the empty file.  
But we're getting far ahead of yourselves now...

## Understanding Git

- The Basics
- The Git Object Database
- History Inspection
- Branching Workflows
- Collaboration
- Advanced Stuff
- Troubleshooting

We'll sortof work our way up towards proficiency starting with the most basic commands, and ending with collaboration.

## The Basics

The most basic tasks you need to be able to do with Git are thus:

- git init
- git status
- git add
- git commit

The very most basic way you would use a git repository as you work records the steps you took to get your repository where it is now.

You initialize it, work a little, use git status to ask git what you've done, "add" some files (or parts of files) to be snapshotted together, and then you tell git you're really serious about it and you want it recorded forever.

## The Basics

The most basic tasks you need to be able to do with Git are thus:

- git init
- git status
- git add
- git commit
- git tag

And we'll throw in git tag, since it'll round out a discussion I'd like to have about the *effect* of all of these commands.

Git tag creates a permanent, human-readable and optionally annotated and signed pointer to some commit, so you'll be able easily to get back to important commits later, such as release points.

```
$>git init
```

```
$>git init  
Initialized empty Git repository in .git/
```

Initializes an empty repo

```
$>git init  
Initialized empty Git repository in .git/  
$>git add .
```

git add tells git that you'll soon want *some* of your recent changes (and perhaps not others) to be recorded for posterity

```
$>git init  
Initialized empty Git repository in .git/  
$>git add .  
$> git status
```

I'll interject here that it's always a good idea to run `git status` before you do something permanent, as a quick check of your grasp of the situation.

```
$>git init
Initialized empty Git repository in .git/
$>git add .
$> git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file: README
#       new file: Rakefile
#       new file: lib/mylib.rb
#
```

And there you go. It even tells you what to do if you don't want one of those files in the next commit.

```
$>git init  
Initialized empty Git repository in .git/  
$>git add .  
$>git commit -m 'my first commit'
```

Git commit takes a snapshot of the way everything is now, reusing what it can, and carves it in stone

```
$>git init
Initialized empty Git repository in .git/
$>git add .
$>git commit -m 'my first commit'
Created initial commit bfe09f9: my first commit
  3 files changed, 3 insertions(+), 0 deletions(-)
  create mode 100644 README
  create mode 100644 Rakefile
  create mode 100644 lib/mylib.rb
$>_
```

And it tells you what it did

```
$>git init  
Initialized empty Git repository in .git/  
$>git add .  
$>git commit -m 'my first commit'  
Created initial commit bfe09f9: my first commit  
 3 files changed, 3 insertions(+), 0 deletions(-)  
 create mode 100644 README  
 create mode 100644 Rakefile  
 create mode 100644 lib/mylib.rb  
$>_
```

Including a shortened SHA of the commit and a summary of the changes it introduces.

```
$>git init
Initialized empty Git repository in .git/
$>git add .
$>git commit -m 'my first commit'
Created initial commit bfe09f9: my first commit
  3 files changed, 3 insertions(+), 0 deletions(-)
  create mode 100644 README
  create mode 100644 Rakefile
  create mode 100644 lib/mylib.rb
$> git tag first_commit
$>
```

And we'll add a tag named "first\_commit" for kicks.

```
➤ git init  
➤ (work work work work)  
➤ git status  
➤ git add --all  
➤ git status  
➤ git commit  
➤ (work work work)  
➤ git status  
➤ git add ...  
➤ git status  
➤ git commit  
➤ git tag  
➤ ...
```



This will be about the order of your normal micro workflow. Lots of working (sorry), adding, committing, and maybe an occasional tag.

# Understanding Git

- ~~The Basics~~
- The Git Object Database
  - History inspection
  - Branching workflows
  - Collaboration
  - Advanced Stuff
  - Troubleshooting

now for a discussion of the git object database, the major component of git's back-end,  
for de-mystification purposes

## The Git Object Database

the Git Object Database is a content-addressable filesystem in the back-end of git

## The Git Object Database

- Key / Value     $\Leftrightarrow$     SHA1 / Content

You give it some arbitrary content, and it stores it, hashes it, and gives you back the 40-char SHA1 hash key you can use to retrieve it again.

# The Git Object Database

- Key / Value   <=>  SHA1 / Content
- All objects go under the “objects” directory under .git

```
.git
├── HEAD
├── config
├── description
├── hooks
│   └── ...
└── info
    └── exclude
└── objects  ← there
    ├── info
    └── pack
└── refs
    ├── heads
    └── tags
```

8 directories, 12 files

all objects go in subdirectories under the objects directory

# The Git Object Database

```
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
│   └── ...
├── index
├── info
│   └── exclude
├── logs
│   └── ...
└── objects
    ├── 54
    │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
    ├── cc
    │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
    ├── e6
    │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
    ├── info
    └── pack
└── refs
    ├── heads
    │   └── master
    └── tags
```

14 directories, 20 files

and when objects are stored, the first two characters of the SHA is the directory, and the remaining characters the filename

## The Git Object Database

- Key / Value    <=>    SHA1 / Content
- All objects go under the “objects” directory under .git
- There are only four kinds of objects that Git stores during normal use:
  - blobs (≈files)
  - trees (≈directories)
  - commits
  - tags

There are only four kinds of objects that Git stores during normal use.  
but first...

## The Git Object Database

*all* git objects are stored as follows

the way *all* objects are stored

## The Git Object Database

**content**

when the object database is given some content to store, it prepends a small header, which is

# The Git Object Database

```
content

new_content = type + ' ' + content.size + \0
+ content
```

## The Git Object Database

```
content

new_content = type + ' ' + content.size + \0
+ content

sha = Digest::SHA1.hexdigest(new_content)
```

it then calculates the sha1 value of  
that new content

## The Git Object Database

```
content

new_content = type + ' ' + content.size + \0
+ content

sha = Digest::SHA1.hexdigest(new_content)
"824aed035c0aa75d64c..."
```

which is how it will refer to it from now on (like an inode number)

## The Git Object Database

```
content

new_content = type + ' ' + content.size + \0
+ content

sha = Digest::SHA1.hexdigest(new_content)
"824aed035c0aa75d64c..."

compressed = zlib::deflate(new_content)
```

then zlib::deflates it,

## The Git Object Database

```
content

new_content = type + ' ' + content.size + \0
+ content

sha = Digest::SHA1.hexdigest(new_content)
      "824aed035c0aa75d64c..."

compressed = zlib::Deflate.compress(new_content)
path = ".git/objects/82/4aed035c0aa75d64c..."
```

determines it's path by the first two characters and the rest of the sha

## The Git Object Database

```
content

new_content = type + ' ' + content.size + \0
+ content

sha = Digest::SHA1.hexdigest(new_content)
"824aed035c0aa75d64c..."

compressed = zlib::deflate(new_content)

path = ".git/objects/82/4aed035c0aa75d64c..."
File.open(path, 'w') {|f| f.write(compressed)}
```

and stores the compressed data in the object database directory.

## The Git Object Database

```
content

new_content = type + ' ' + content.size + '\0'
+ content
sha = Digest::SHA1.hexdigest(new_content)
      "824aed035c0aa75d64c..."

compressed = zlib::deflate(new_content)

path = ".git/objects/82/4aed035c0aa75d64c..."

File.open(path, 'w') {|f| f.write(compressed)}
```

### “loose” format

this is referred to as the “loose” format

## The Git Object Database

`git gc`

objects occasionally get stored  
another way  
Every once in a while, the garbage  
collector is run automatically.

## The Git Object Database

`git gc`

same file with minor differences

Many housekeeping tasks are done,

## The Git Object Database

```
git gc
```

same file with minor differences

```
.git/objects/82/4aed035c0aa75d64c...
.git/objects/1d/c9cbb76cbb80fce1...
.git/objects/63/874f37013c1740acd...
.git/objects/04/fb8aee105e6e445e8...
.git/objects/45/b983be36b73c0788d...
.git/objects/f1/032eed02413a1145c...
```

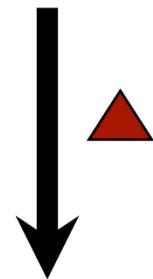
one of which is that whenever git can detect that a bunch of snapshots are of the same file with minor differences,

## The Git Object Database

`git gc`

same file with minor differences

```
.git/objects/82/4aed035c0aa75d64c...
.git/objects/1d/c9cbb76cbb80fce1...
.git/objects/63/874f37013c1740acd...
.git/objects/04/fb8aee105e6e445e8...
.git/objects/45/b983be36b73c0788d...
.git/objects/f1/032eed02413a1145c...
```



```
.git/objects/pack/pack-999727..9f600.pack
.git/objects/pack/pack-999727..9f600.idx
```

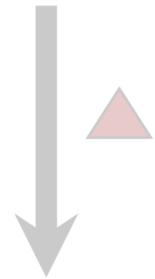
It will collect and delta-compress them into a “pack” file and index.

## The Git Object Database

`git gc`

same file with minor differences

```
.git/objects/82/4aed035c0aa75d64c...
.git/objects/1d/c9cbcb76cbb80fce1...
.git/objects/63/874f37013c1740acd...
.git/objects/04/fb8aee105e6e445e8...
.git/objects/45/b983be36b73c0788d...
.git/objects/f1/032eed02413a1145c...
```



```
.git/objects/pack/pack-999727..9f600.pack
.git/objects/pack/pack-999727..9f600.idx
```

These are stored separately from the loose objects, in `.git/objects/pack`.

## The Git Object Database

git gc

same file with minor differences

.git/objects/82/14aed01c0a75d64c...  
“packed” format  
.git/objects/00/00000000000000000000000000000000...  
.git/objects/63/874f37013c1740acd...  
.git/objects/04/fb8aee105e6e445e8...  
.git/objects/45/b983be36b73c0788d...  
.git/objects/f1/032eed02413a1145c...  
  
.git/objects/pack/pack-999727..9f600.pack  
.git/objects/pack/pack-999727..9f600.idx



As you've probably guessed, this is called the “packed” format.

## The Git Object Database

There are only four types of Git objects

now back to this  
there are only four types of git objects  
that can be stored in the database

# The Git Object Database

blob

the blob

## The Git Object Database

blob

tree

the tree

## The Git Object Database

blob

tree

commit

the commit

## The Git Object Database

blob

tree

commit

tag

and the tag. first, we'll cover

## The Git Object Database

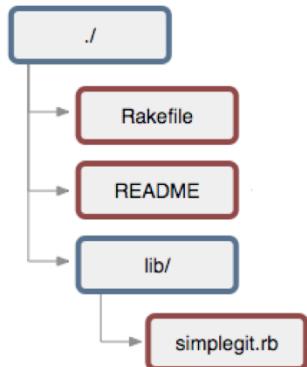
blob

the blob

# The Git Object Database

blob

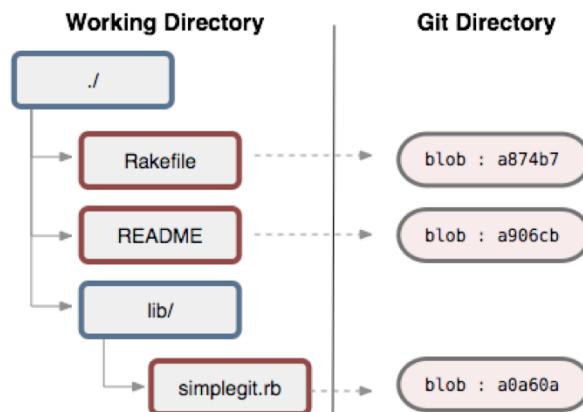
## Working Directory



If you have a working directory that looks like this,

## The Git Object Database

blob



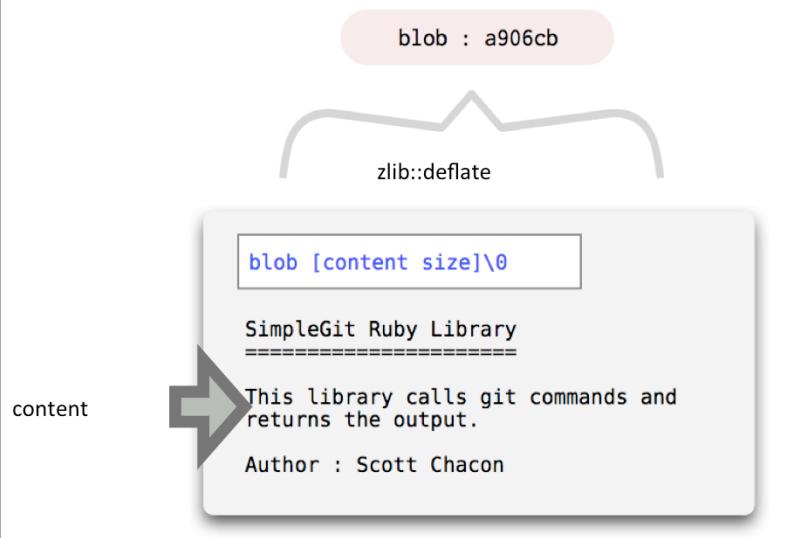
Then the blobs are the git object database equivalent of the files.

# The Git Object Database



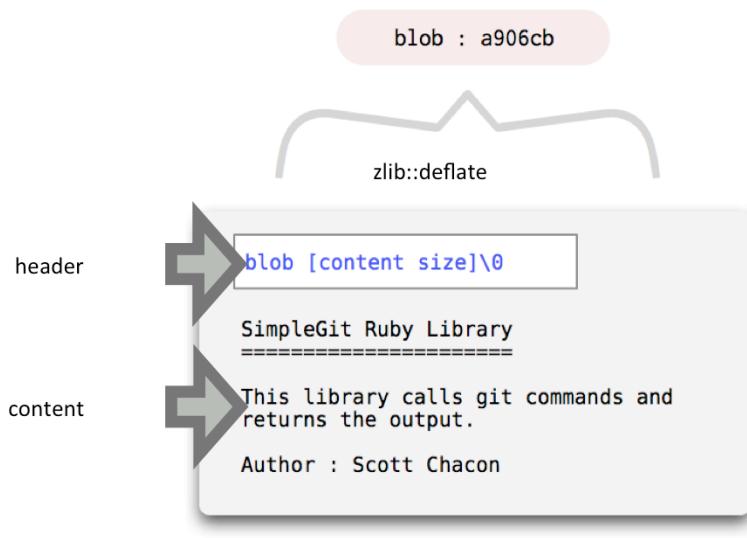
So say you have a little README file in a project called “SimpleGit”, and you’ve committed it to the repository.

# The Git Object Database



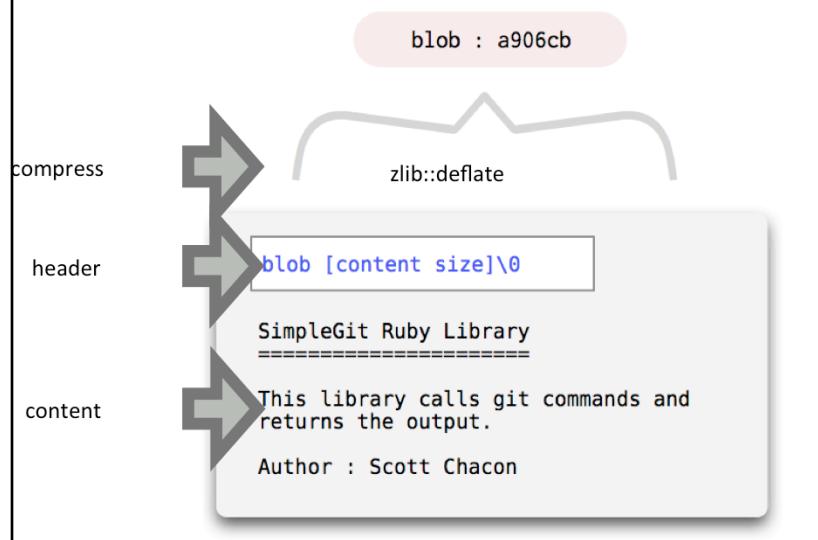
The content is simply text,

# The Git Object Database



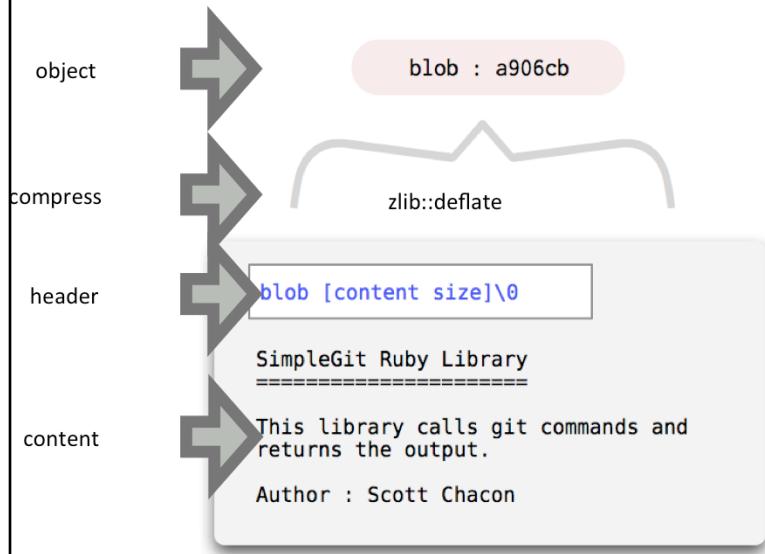
As mentioned before the header has the string “blob”, a space, the content size, and a null-byte,

# The Git Object Database



It's compressed,

## The Git Object Database



and handed to the Git Object Database, where it's assigned a SHA.

## The Git Object Database

blob

tree

commit

tag

Next up,

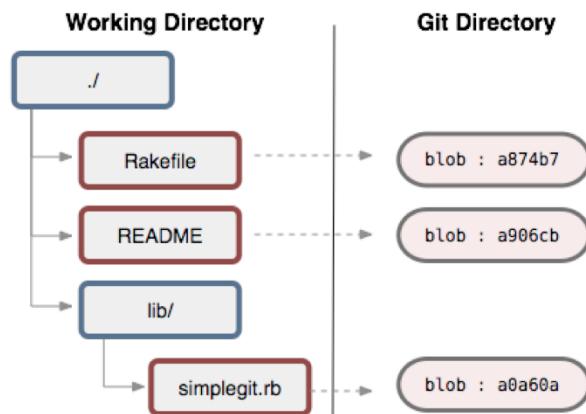
## The Git Object Database

tree

Trees

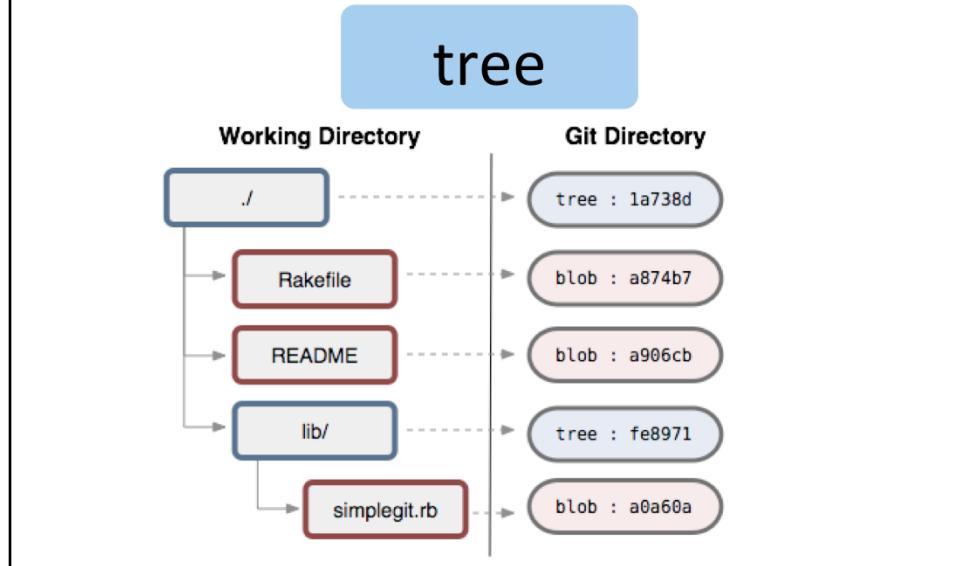
## The Git Object Database

tree



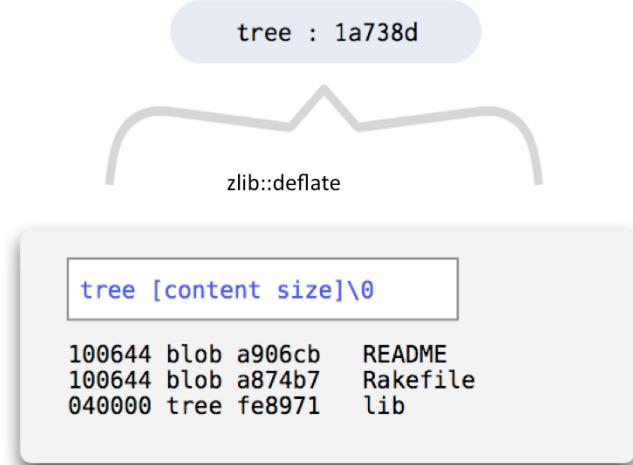
if the files correspond to “blob”s

## The Git Object Database



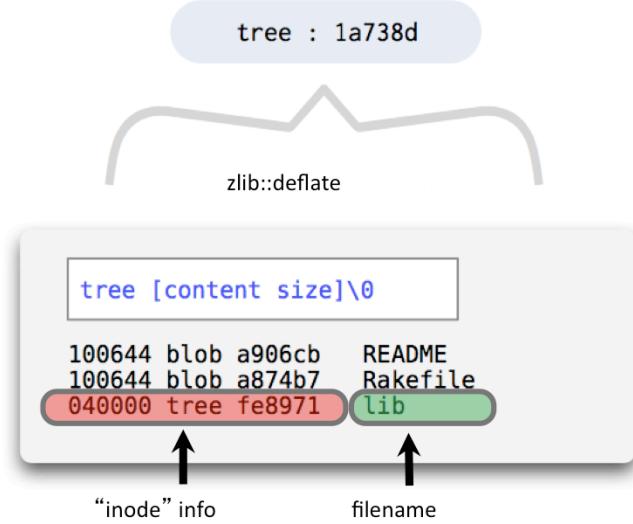
the directories are “trees”

## The Git Object Database



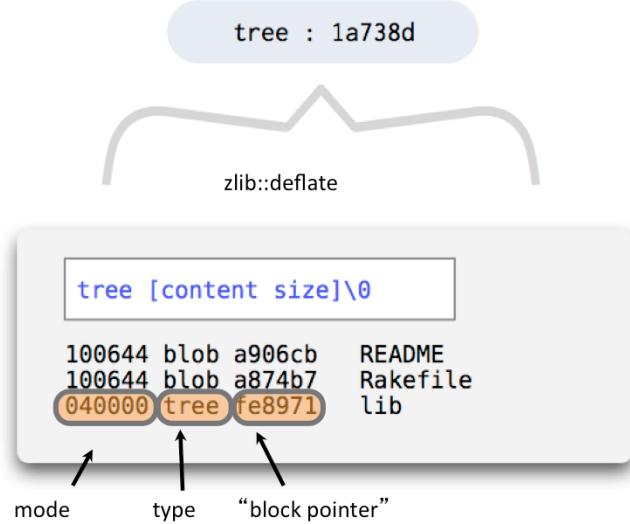
this. for you posix geeks, they are sort of a combination of

## The Git Object Database



the posix directory data

## The Git Object Database



and a subset of the inode information itself, smushed together into one object

## The Git Object Database

blob

tree

commit

tag

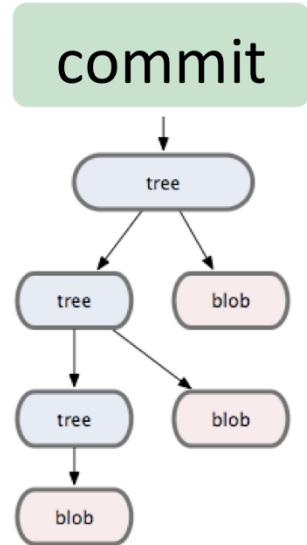
now, the third object is the “commit”

## The Git Object Database

commit

now, the third object is the “commit”

## The Git Object Database



a commit object points to the root tree  
of your project

## The Git Object Database



and stores a bunch of meta-information about that particular snapshot of content, including

# The Git Object Database



the commit message

# The Git Object Database



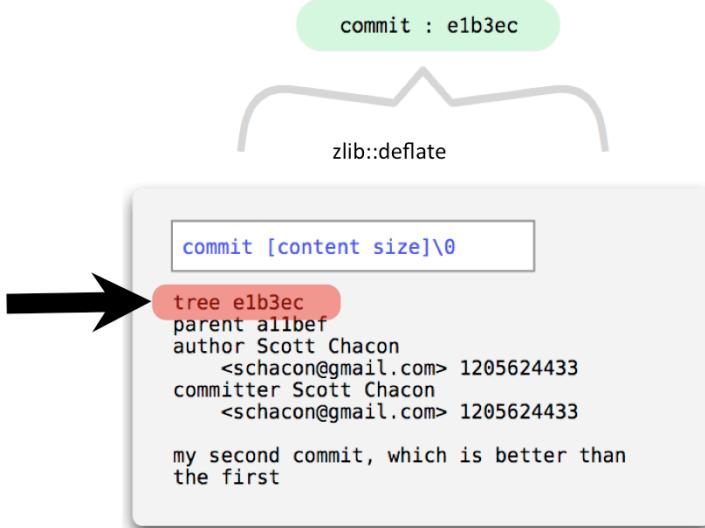
the author and committer and the date-times it was authored and committed

## The Git Object Database



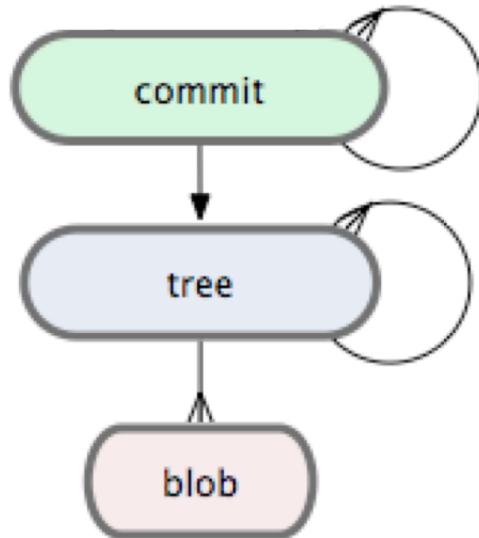
a pointer to the previous commit or commits (if it is a merge)

## The Git Object Database



and a pointer to the root tree of the project

## The Git Object Database



which gives us a directed acyclic graph like this

## The Git Object Database

blob

tree

commit

tag

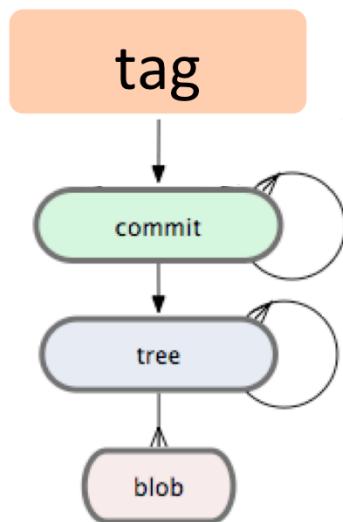
no now, we come to the last object,  
the “tag”

## The Git Object Database

tag

no now, we come to the last object,  
the “tag”

## The Git Object Database



a tag object points to a commit, to provide a way to reference a commit object by a human readable string, rather than a sha value

# The Git Object Database



the commit object contains

# The Git Object Database



a tag message

# The Git Object Database



a tagger and date

# The Git Object Database



a tag string

## The Git Object Database



and the object you're tagging, which is normally a commit

```
$> git init  
$> touch README  
$> git add .  
$> git commit -m  
"initial commit"  
  
$> tree -a .git
```

```
.git  
├── COMMIT_EDITMSG  
├── HEAD  
├── config  
├── description  
├── hooks  
│   ├── ...  
├── index  
├── info  
│   └── exclude  
├── logs  
│   ├── ...  
├── objects  
│   ├── 54  
│   │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd  
│   ├── cc  
│   │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52  
│   ├── e6  
│   │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391  
│   ├── info  
│   └── pack  
└── refs  
    ├── heads  
    │   └── master  
    └── tags
```

```
14 directories, 20 files
```

## The Git Object Database

Now I intend to rapidly reconnect all this theoretical information to our concrete repository.

# The Git Object Database

```
$> git init  
$> touch README  
$> git add .  
$> git commit -m  
"initial commit"  
  
$> git tag -a taggy  
(opens editor to  
write annotation)  
  
$> tree -a .git
```

```
.git  
├── COMMIT_EDITMSG  
├── HEAD  
├── config  
├── description  
├── hooks  
│   ├── ...  
├── index  
├── info  
│   └── exclude  
├── logs  
│   ├── ...  
├── objects  
│   ├── 54  
│   │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd  
|   └── c7  
|       └── dcf9ef54b124542e958c62866d8724471d77d2  
|           ├── cc  
|           │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52  
|           └── e6  
|               └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391  
|           └── info  
|           └── pack  
└── refs  
    ├── heads  
    |   └── master  
    └── tags  
        └── taggy
```

15 directories 22 files

First we'll add an annotated tag, which adds a new object to the object database.

```
$> git init  
$> touch README  
$> git add .  
$> git commit -m  
"initial commit"  
  
$> git tag -a taggy  
(opens editor to  
write annotation)  
  
$> tree -a .git
```

```
.git  
├── COMMIT_EDITMSG  
├── HEAD  
├── config  
├── description  
├── hooks  
│   ├── ...  
├── index  
├── info  
│   └── exclude  
├── logs  
│   ├── ...  
└── objects  
    ├── 54  
    │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd  
    ├── c7  
    │   └── dcf9ef54b124542e958c62866d8724471d77d2  
    ├── cc  
    │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52  
    ├── e6  
    │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391  
    └── info  
        └── pack  
└── refs  
    ├── heads  
    │   └── master  
    └── tags  
        └── taggy
```

15 directories 22 files

## The Git Object Database

Now we'll peek into each of these objects to give you an intuition about what git is doing for you with these commands.

# The Git Object Database

```
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
│   ├── ...
├── index
├── info
│   └── exclude
├── logs
│   ├── ...
└── objects
    ├── 54
    │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
    ├── c7
    │   └── dcf9ef54b124542e958c62866d8724471d77d2
    ├── cc
    │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
    ├── e6
    │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
    ├── info
    └── pack
└── refs
    ├── heads
    │   └── master
    └── tags
        └── taggy
```

15 directories 22 files

## The Git Object Database

```
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
│   └── ...
├── index
├── info
│   └── exclude
└── logs
    └── ...
├── objects
    ├── 54
    │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
    └── c7
        └── dcf9ef54b124542e958c62866d8724471d77d2
            ├── cc
            │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
            ├── e6
            │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
            ├── info
            └── pack
└── refs
    ├── heads
    │   └── master
    └── tags
        └── taggy
```

\$> git cat-file -p c7dcf9ef54b124542e958c62866d8724471d77d2  
object cc9fc8c4ea4df4f245103cbe80c35bfa2eb07e52  
type commit  
tag taggy  
tagger Daniel Cox <daniel.cox@six3systems.com> Sat Jan 5  
23:47:45 2013 -0500  
  
a thrilling annotation message  
\$>

the tag pointing to the commit

15 directories 22 files

cat-file -p is a plumbing command that allows you to peek into a commit given its SHA.

It just so happens that the object I asked to see was the tag, which shows what object it points to, some meta information, and the annotation message, as discussed earlier.

## The Git Object Database

```
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
│   ├── ...
│   └── index
├── info
│   └── exclude
└── logs
    ├── ...
    └── objects
        ├── 54
        │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
        ├── c7
        │   └── dcf9ef54b124542e958c62866d8724471d77d2
        ├── cc
        │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
        ├── e6
        │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
        ├── info
        └── pack
└── refs
    ├── heads
    │   └── master
    └── tags
        └── taggy
```

```
$> git cat-file -p c7dcf9ef54b124542e958c62866d8724471d77d2
object cc9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
type commit
tag taggy
tagger Daniel Cox <daniel.cox@six3systems.com> Sat Jan 5
23:47:45 2013 -0500
a thrilling annotation message
$>
```

the tag pointing to the commit

15 directories 22 files

Now we'll open the object it points to, which *should* be the commit...

## The Git Object Database

```
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
│   └── ...
├── index
├── info
│   └── exclude
├── logs
│   └── ...
└── objects
    ├── 54
    │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
    ├── c7
    │   └── dcf9ef54b124542e958c62866d8724471d77d2
    ├── cc
    │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
    ├── e6
    │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
    ├── info
    └── pack
└── refs
    ├── heads
    │   └── master
    └── tags
        └── taggy
```

```
$> git cat-file -p cc9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
```

```
15 directories 22 files
```

## The Git Object Database

```
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
|   ├── ...
|   └── index
|       └── info
|           └── exclude
└── logs
    ├── ...
    └── objects
        ├── 54
        |   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
        |   └── c7
        |       └── dcf9ef54b124542e958c62866d8724471d77d2
        |   └── cc
        |       └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
        |   └── e6
        |       └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
        └── info
        └── pack
└── refs
    ├── heads
    |   └── master
    └── tags
        └── taggy
```

```
$> git cat-file -p cc9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
tree 543b9bebdc6bd5c4b22136034a95dd097a57d3dd
author Daniel Cox <daniel.cox@six3systems.com> 1357417822 -0500
committer Daniel Cox <daniel.cox@six3systems.com> 1357417822
-0500

init
$>
```

the commit pointing to the tree

It is.

The commit contains the SHA of the tree *it* points to, some meta information, and the commit message.

## The Git Object Database

```
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
|   ├── ...
|   └── index
|       └── info
|           └── exclude
└── logs
    ├── ...
└── objects
    ├── 54
    |   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
    ├── c7
    |   └── dcf9ef54b124542e958c62866d8724471d77d2
    ├── cc
    |   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
    ├── e6
    |   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
    ├── info
    └── pack
└── refs
    ├── heads
    |   └── master
    └── tags
        └── taggy
```

```
$> git cat-file -p cc9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
tree 543b9bebdc6bd5c4b22136034a95dd097a57d3dd
author Daniel Cox <daniel.cox@six3systems.com> 1357417822 -0500
committer Daniel Cox <daniel.cox@six3systems.com> 1357417822
-0500

init
$>
```

the commit pointing to the tree

15 directories 22 files

In any commit but the first one, the commit would have a “parent” line, pointing to the SHA of the commit that came immediately before it, here.

## The Git Object Database

```
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
|   ├── ...
|   └── index
|       └── info
|           └── exclude
└── logs
    ├── ...
    └── objects
        ├── 54
        |   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
        ├── c7
        |   └── dcf9ef54b124542e958c62866d8724471d77d2
        ├── cc
        |   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
        ├── e6
        |   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
        ├── info
        └── pack
└── refs
    ├── heads
    |   └── master
    └── tags
        └── taggy
```

```
$> git cat-file -p cc9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
tree 543b9bebdc6bd5c4b22136034a95dd097a57d3dd
author Daniel Cox <daniel.cox@six3systems.com> 1357417822 -0500
committer Daniel Cox <daniel.cox@six3systems.com> 1357417822
-0500

init
$>
```

the commit pointing to the tree

15 directories 22 files

Let's open the tree this commit points to next.

## The Git Object Database

```
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
│   └── ...
├── index
├── info
│   └── exclude
├── logs
│   └── ...
└── objects
    ├── 54
    │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
    ├── c7
    │   └── dcf9ef54b124542e958c62866d8724471d77d2
    ├── cc
    │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
    ├── e6
    │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
    ├── info
    └── pack
└── refs
    ├── heads
    │   └── master
    └── tags
        └── taggy
```

```
$> git cat-file -p 543b9bebdc6bd5c4b22136034a95dd097a57d3dd
```

```
15 directories 22 files
```

## The Git Object Database

```
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
│   └── ...
├── index
├── info
│   └── exclude
├── logs
│   └── ...
└── objects
    ├── 54
    │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
    ├── c7
    │   └── dcf9ef54b124542e958c62866d8724471d77d2
    ├── cc
    │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
    ├── e6
    │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
    ├── info
    └── pack
└── refs
    ├── heads
    │   └── master
    └── tags
        └── taggy
```

```
$> git cat-file -p 543b9bebdc6bd5c4b22136034a95dd097a57d3dd
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 README
$>
```

the tree with one file in it

15 directories 22 files

The tree contains one blob, and when that blob is checked out into the working directory, it will be named README.

## The Git Object Database

```
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
│   └── ...
├── index
├── info
│   └── exclude
├── logs
│   └── ...
└── objects
    ├── 54
    │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
    ├── c7
    │   └── dcf9ef54b124542e958c62866d8724471d77d2
    ├── cc
    │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
    ├── e6
    │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
    ├── info
    └── pack
└── refs
    ├── heads
    │   └── master
    └── tags
        └── taggy
```

```
$> git cat-file -p e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

```
$>
```

the empty file blob

And here's that empty blob.

# Understanding Git

- ~~The Basics~~
- ~~The Git Object Database~~
- History inspection
- Branching workflows
- Collaboration
- Advanced Stuff
- Troubleshooting

## History Inspection

History Inspection.

So, now you know how to record the history of your project as you work, and what git is doing for you, but for that information to be useful we'll need to be able to inspect the history we've created.

## History Inspection

- `git log`
  - shows you a nice ordered list of your commits and their messages
- `git diff`
  - allows you to see the difference between two versions of a file, or between two entire commits
- `git show`
  - ask to see something git is holding for you. commits, trees, tags, blobs, and any *treeish*

so we'll add three new commands, `git log`, `diff` and `show`

git log

```
$>git log_
```

git log, without any arguments

```
$>git log
commit 310154e3c7db47d8bac935c2c43aee6afac11aae
Author: Scott Chacon <schacon@gmail.com>
Date:   Sun Apr 13 10:49:15 2008 -0700

    updated README formatting and added blame

commit f7f3f6dd8fd3fa40f052427c32785a0fa01aaa5f
Author: Magnus Chacon <mchacon@gmail.com>
Date:   Sun Apr 13 10:45:01 2008 -0700

    changed my name a bit

commit 710f0f8d2cdf5af87033b9ec08859a505f9a6af5
Author: Magnus Chacon <mchacon@gmail.com>
Date:   Sun Apr 13 10:34:16 2008 -0700

    added ls-files

commit c110d7ff8cfb86fd5cce9a8aee462678dbb4ef9b
Author: Scott Chacon <schacon@gmail.com>
Date:   Sun Apr  6 12:13:36 2008 -0700

    made the ls-tree function recursive and list trees
```

prints out a pretty nice history of your commits in reverse order

```
$>git log --pretty=oneline
```

or, there are many options you can pass to reformat the logs or get more information from them

```
$>git log --pretty=oneline
3e625e2139a71bfbaa85b055c139bceafab7e2c0 updated rakefile
4fefbaafd51ff3e79b84e9ee1a0dac2848e21a98 commit the todo
c4f8fd68ae3897d1d10456ed2b93d87cdc53d6e8 updated README
a5f4a0daa9b13bb85283461ddaba6e589e34ccb5 added cat-file
310154e3c7db47d8bac935c2c43aee6afac11aae updated README formatting
f7f3f6dd8fd3fa40f052427c32785a0fa01aaa5f changed my name a bit
710f0f8d2cdf5af87033b9ec08859a505f9a6af5 added ls-files
c110d7ff8cfb86fd5cce9a8aee462678dbb4ef9b made the ls-tree function
ce9b0d5551762048735dd67917046b44176317e0 limiting log to 30
e22e01e39c39521edc8cccd3aa5df228209e7ad2d -added todo options
2c0d4d7e3d36a8cd5da1eab3e0c1e83faa3bfd9f added limit to log functio
47c668a9e8f93f65ff9019c587f3b2204bd6c98d fixed conflict
ca28a28e64192af4527ea63665dceaa9bdb6a4b new version 0.1.2
f30ba8480d9aba88580696651e80d9aabe8e15a2 new version 0.2.0
1edee6b1d61823a2de3b09c160d7080b8d1b3a40 added a new function
4b0780c5b7a4ae1217f4da742a172e7d787ca3eb rakefile and todo file add
0d57a4eb03d35ele1041857d1cb01199eeb23cc4 Revert "rakefile and todo
f65a297eb0ace63e5b302e849064f37ded909c0b rakefile and todo file add
c035c611b559537d725d875cfa10c05c17d48e4f Merge branch 'newfunc'
d6fad7dd6f98481cab874b192aaa79d03490d4a0 added more description
1a8c32ea9725886aaadc35dbf4d68468d3af0abe added lstree function
40e5c10f67fa01c01da5687d08bd3bc21bb6a44b added email to readme
cf25cc3bfb0ece7dc3609b8dc0cc4ale19ffbcd4 committing all changes
0c8a9ec46029a4e92a428cb98c9693f09f69a3ff changed the verison number
```

```
$>git log --pretty=format:"%h %an %ar - %s" _
```

you can also make up your own format

```
$>git log --pretty=format:"%h %an %ar - %s"
3e625e2 Magnus Chacon 47 minutes ago - updated rakefile
4fefbaa Magnus Chacon 56 minutes ago - commit the todo
c4f8fd6 Magnus Chacon 60 minutes ago - updated README
a5f4a0d Magnus Chacon 2 weeks ago - added cat-file
310154e Scott Chacon 2 weeks ago - updated README formatting and a
f7f3f6d Magnus Chacon 2 weeks ago - changed my name a bit
710f0f8 Magnus Chacon 2 weeks ago - added ls-files
c110d7f Scott Chacon 3 weeks ago - made the ls-tree function recur
ce9b0d5 Scott Chacon 3 weeks ago - limiting log to 30
e22e01e Scott Chacon 3 weeks ago - -added todo options
2c0d4d7 Scott Chacon 3 weeks ago - added limit to log function
47c668a Scott Chacon 3 weeks ago - fixed conflict
ca28a28 Scott Chacon 3 weeks ago - new version 0.1.2
f30ba84 Scott Chacon 3 weeks ago - new version 0.2.0
1edee6b Scott Chacon 3 weeks ago - added a new function
4b0780c Scott Chacon 3 weeks ago - rakefile and todo file added
0d57a4e Scott Chacon 3 weeks ago - Revert "rakefile and todo file
f65a297 Scott Chacon 3 weeks ago - rakefile and todo file added
c035c61 Scott Chacon 3 weeks ago - Merge branch 'newfunc'
d6fad7d Scott Chacon 3 weeks ago - added more description
1a8c32e Scott Chacon 3 weeks ago - added lstree function
40e5c10 Scott Chacon 3 weeks ago - added email to readme
cf25cc3 Scott Chacon 6 weeks ago - committing all changes
0c8a9ec Scott Chacon 6 weeks ago - changed the verision number
0576faa Scott Chacon 6 weeks ago my second commit which is bot+
```

```
$>git log --pretty=format:"%h %an %ar - %s"
3e625e2 Magnus Chacon 47 minutes ago - updated rakefile
4fefbaa Magnus Chacon 56 minutes ago - commit the todo
c4f8fd6 Magnus Chacon 60 minutes ago - updated README
a5f4a0d Magnus Chacon 2 weeks ago - added cat-file
310154e Scott Chacon 2 weeks ago - updated README formatting and a
f7f3f6d Magnus Chacon 2 weeks ago - changed my name a bit
710f0f8 Magnus Chacon 2 weeks ago - added ls-files
c110d7f Scott Chacon 3 weeks ago - made the ls-tree function recur
ce9b0d5 Scott Chacon 3 weeks ago - limiting log to 30
e22e01e Scott Chacon 3 weeks ago - -added todo options
2c0d4d7 Scott Chacon 3 weeks ago - added limit to log function
47c668a Scott Chacon 3 weeks ago - fixed conflict
ca28a28 Scott Chacon 3 weeks ago - new version 0.1.2
f30ba84 Scott Chacon 3 weeks ago - new version 0.2.0
1edee6b Scott Chacon 3 weeks ago - added a new function
4b0780c Scott Chacon 3 weeks ago - rakefile and todo file added
0d57a4e Scott Chacon 3 weeks ago - Revert "rakefile and todo file
f65a297 Scott Chacon 3 weeks ago - rakefile and todo file added
c035c61 Scott Chacon 3 weeks ago - Merge branch 'newfunc'
d6fad7d Scott Chacon 3 weeks ago - added more description
1a8c32e Scott Chacon 3 weeks ago - added lstree function
40e5c10 Scott Chacon 3 weeks ago - added email to readme
cf25cc3 Scott Chacon 6 weeks ago - committing all changes
0c8a9ec Scott Chacon 6 weeks ago - changed the verision number
0576faa Scott Chacon 6 weeks ago - my second commit which is bot+
```

in this one we have a shortened SHA,

```
$>git log --pretty=format:"%h %an %ar - %s"
3e625e2 Magnus Chacon 47 minutes ago - updated rakefile
4fefbaa Magnus Chacon 56 minutes ago - commit the todo
c4f8fd6 Magnus Chacon 60 minutes ago - updated README
a5f4a0d Magnus Chacon 2 weeks ago - added cat-file
310154e Scott Chacon 2 weeks ago - updated README formatting and a
f7f3f6d Magnus Chacon 2 weeks ago - changed my name a bit
710f0f8 Magnus Chacon 2 weeks ago - added ls-files
c110d7f Scott Chacon 3 weeks ago - made the ls-tree function recur
ce9b0d5 Scott Chacon 3 weeks ago - limiting log to 30
e22e01e Scott Chacon 3 weeks ago - -added todo options
2c0d4d7 Scott Chacon 3 weeks ago - added limit to log function
47c668a Scott Chacon 3 weeks ago - fixed conflict
ca28a28 Scott Chacon 3 weeks ago - new version 0.1.2
f30ba84 Scott Chacon 3 weeks ago - new version 0.2.0
1edee6b Scott Chacon 3 weeks ago - added a new function
4b0780c Scott Chacon 3 weeks ago - rakefile and todo file added
0d57a4e Scott Chacon 3 weeks ago - Revert "rakefile and todo file
f65a297 Scott Chacon 3 weeks ago - rakefile and todo file added
c035c61 Scott Chacon 3 weeks ago - Merge branch 'newfunc'
d6fad7d Scott Chacon 3 weeks ago - added more description
1a8c32e Scott Chacon 3 weeks ago - added lstree function
40e5c10 Scott Chacon 3 weeks ago - added email to readme
cf25cc3 Scott Chacon 6 weeks ago - committing all changes
0c8a9ec Scott Chacon 6 weeks ago - changed the verision number
0576faa Scott Chacon 6 weeks ago - my second commit which is bot+
```

the commit author, etc.



git diff

git diff

## git diff (treeish1) (treeish2)

```
diff --git a/.gitignore b/.gitignore
index 224f138..7e56ec7 100644
--- a/.gitignore
+++ b/.gitignore
@@ -1,4 +1,3 @@
 bin/*.*log
 pkg
 coverage
-ticgit*gem
diff --git a/Rakefile b/Rakefile
index b30e89c..bd7b91a 100644
--- a/Rakefile
+++ b/Rakefile
@@ -31,7 +31,6 @@ desc "Clean out the coverage and pkg directories"
task :clean do
  rm_rf 'coverage'
  rm_rf 'pkg'
- rm Dir.glob('ticgit*gem')
end

task :default => "pkg/#{spec.name}-#{spec.version}.gem" do
diff --git a/ticgit.gemspec b/ticgit.gemspec
index c0e96b7..775eaef1 100644
--- a/ticgit.gemspec
+++ b/ticgit.gemspec
@@ -6,7 +6,7 @@ Gem::Specification.new do |s|
  s.author    = "Scott Chacon"
  s.email     = "schacon@gmail.com"
  s.summary   = "A distributed ticketing system for Git projects."
- s.files    = ["lib/ticgit/base.rb", "lib/ticgit/cli.rb", "lib/ticgit/comment.rb"
+ s.files    = ["lib/ticgit", "lib/ticgit/base.rb", "lib/ticgit/cli.rb", "lib/ticg
  s.bindir   = 'bin'
  s.executables << "tic"
```

git diff will actually give you a nice unified diff of two files or of states of your repository, or of the changes you've made since you last git-added anything... But what's a treeish?

the treeish

## the treeish

alternate ways to refer to  
objects or ranges of objects

they're a big bag of different ways to refer to a commit or commits ...  
(*which* you should remember are just snapshots of the way your working directory is  
or was)

## Treeish

- full sha-1
- partial sha-1
- branch or tag name
- date spec
- ordinal spec
- carrot parent
- tilde spec
- tree pointer
- blob spec
- ranges

## Full SHA1

6e453f523fa1da50ecb04431101112b3611c6a4d

## Partial SHA1

6e453f523fa1da50ecb04431101112b3611c6a4d

6e453f523fa1da50

6e453

Branch, Remote  
or Tag Name

v1.0

master

origin/testing

## Date Spec

```
master@{yesterday}  
master@{1 month ago}
```

## Ordinal Spec

`master@{5}`

5th prior value of ‘master’

## Carrot Parent

`master^2`

2nd parent of ‘master’

This one will make sense later when we talk about merges, and thus commits that can have multiple parent commits.

# Tilde Spec

**master~2**

2nd generation grandparent of ‘master’

## Tree Pointer

`master^ {tree}`

tree that ‘master’ points to

## Blob Spec

`master:/path/to/file`

blob of that file in ‘master’ commit

# Ranges

`ce0e4..e4272`

everything between two commits

# Ranges

`ce0e4..`

everything since a commit

## One Especially Useful One

`origin/master..HEAD`

or just

`origin/master..`

everything you've done since the last time you pulled

to get ahead of ourselves a little bit, here's one especially useful range which will allow you to preview what you're about to push  
use with git log, or with git diff if you need a bit more detail

## History Inspection

- `git log`
- `git diff`
- `git show`

`git show`.

`git show` will try to show you something useful for whatever treeish you give it.

## History Inspection

- `git log`
- `git diff`
- `git show`

```
$> git show taggy
tag taggy
Tagger: Daniel Cox <daniel.cox@six3systems.com>
Date:   Sat Jan 5 23:47:45 2013 -0500

a thrilling annotation message

commit cc9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
Author: Daniel Cox <daniel.cox@six3systems.com>
Date:   Sat Jan 5 15:30:22 2013 -0500

init

diff --git a/README b/README
new file mode 100644
index 000000..e69de29
```

So if you give it a tag, such as the one we made earlier, we get the tag information, the information for the commit it points to, and a diff of what that commit introduces.

# Understanding Git

- The Basics
- The Git Object Database
- History inspection
- Branching workflows
- Collaboration
- Advanced Stuff
- Troubleshooting

## Branching Workflows

Now we control our present and remember our past, but what of the future.  
Of course we'll be working on several things at once, and what we're working on now  
will probably be different than what we have in production, or even what our  
colleagues are basing their work off of.  
We need the concept of branches to make that easier, and so git supplies very cheap  
branching

## Branching Workflows

- git branch
- git checkout
- git merge
- git rebase

## Branching Workflows

- `git branch`
  - list, create or delete branches

`git branch` lists, creates or deletes branches

## Branching Workflows

- git branch
  - create

```
$> git branch new_branch  
$>
```

here's the creation of a new branch

## Branching Workflows

- `git branch`
  - list,

```
$> git branch
* master
  new_branch
```

the listing of the branches, with a little asterisk there to show you which branch you're currently “on”

# Branching Workflows

- `git branch`
  - `delete`

Gentle

```
$> git branch -d new_branch  
$>
```

Drastic

```
$> git branch -D new_branch  
$>
```

and deletion of branches

little-d delete will only successfully delete the branch you name if the branch you're *on* already "contains" all the same commits.

big-D delete nukes the branch no matter what, which is less drastic than it might sound, since the actual orphaned commits aren't actually removed for months.  
deleting a branch only deletes the branch *reference*.

## Branch Reference?

What's a branch reference?

Branch Reference?

blob

tree

commit

tag

You'll recall that there are only four types of objects in the object database.

No Branch References Here

blob

tree

**immutable**

commit

tag

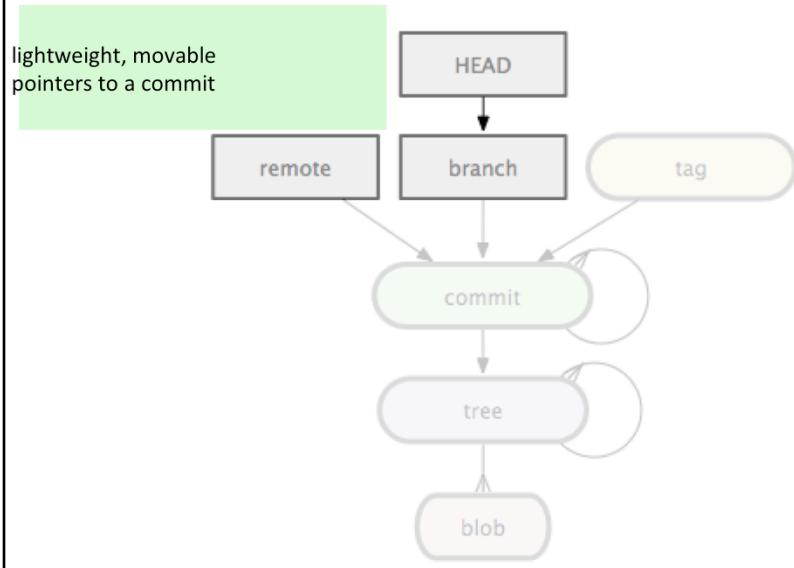
All of these objects are immutable.  
That's a problem, since the closest  
thing we have so far to a named  
branch reference is a tag,

## No Branch References Here



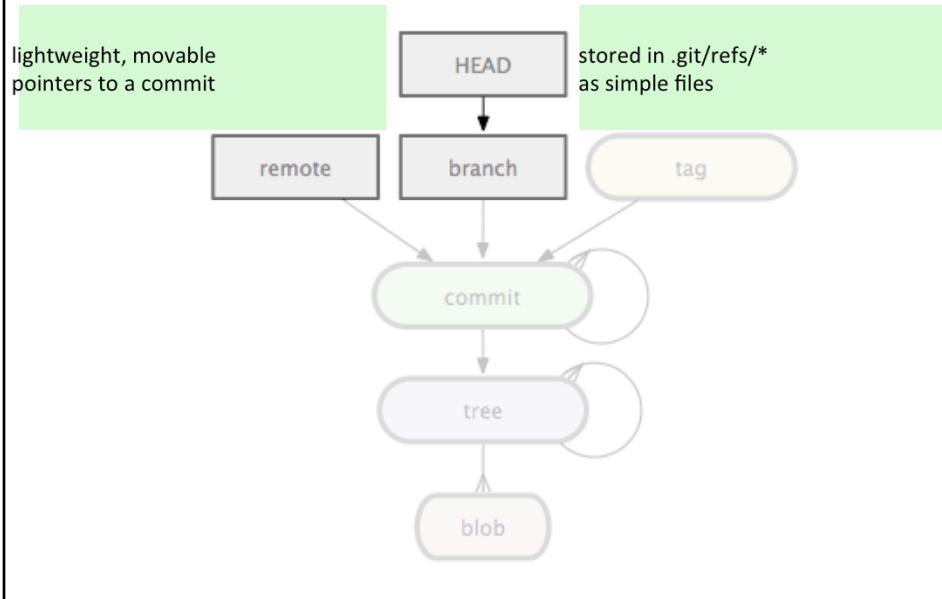
and when we're “on” a particular branch... and we *commit*, clearly we want the branch to change and point to the latest commit.  
we need a named pointer to the *last* commit on an ever-growing *branch* of commits

## References



so git has *references*, which are lightweight, easily movable pointers to a commit that signifies the head of a local or remote branch

## References



they're stored in `.git/refs`

# References

```
.git
├── COMMIT_EDITMSG
└── HEAD
    └── ...
├── config
├── description
├── hooks
│   ├── ...
├── index
├── info
│   └── exclude
├── logs
│   ├── ...
├── objects
│   ├── 54
│   │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
│   ├── c7
│   │   └── dcf9ef54b124542e958c62866d8724471d77d2
│   ├── cc
│   │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
│   ├── e6
│   │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
│   ├── info
│   └── pack
└── refs
    ├── heads
    │   ├── master
    │   └── new_branch
    └── tags
        └── taggy
```

# References

```
.git
├── COMMIT_EDITMSG
└── HEAD
├── config
├── description
├── hooks
│   ├── ...
├── index
├── info
│   └── exclude
├── logs
│   ├── ...
├── objects
│   ├── 54
│   │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
│   ├── c7
│   │   └── dcf9ef54b124542e958c62866d8724471d77d2
│   ├── cc
│   │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
│   ├── e6
│   │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
│   ├── info
│   └── pack
└── refs
    ├── heads
    │   ├── master
    │   └── new_branch
    └── tags
        └── taggy
```

← Which branch you're "on"

← Branch References!

## References

```
.git
├── COMMIT_EDITMSG
└── HEAD
├── config
├── description
├── hooks
│   └── ...
├── index
├── info
│   └── exclude
├── logs
│   └── ...
├── objects
│   ├── 54
│   │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
│   ├── c7
│   │   └── dcf9ef54b124542e958c62866d8724471d77d2
│   ├── cc
│   │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
│   ├── e6
│   │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
│   ├── info
│   └── pack
└── refs
    ├── heads
    │   ├── master
    │   └── new_branch
    └── tags
        └── taggy
```

← Which branch you're "on"

```
$> cat .git/HEAD
ref: refs/heads/master
```

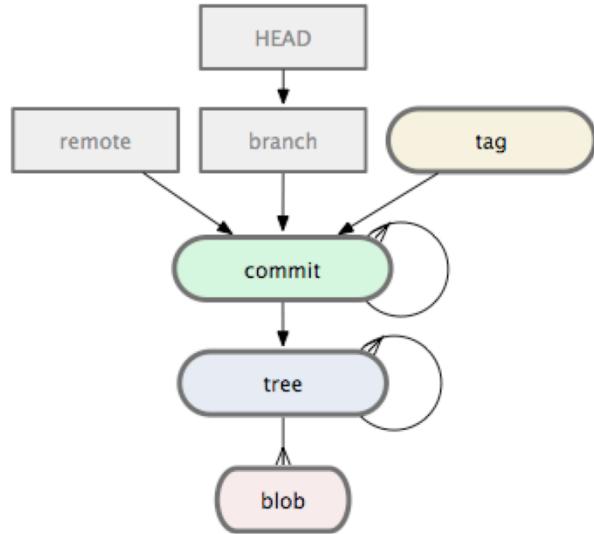
← Branch References!

```
$> cat .git/refs/heads/master
cc9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
```

and this is important: the references themselves contain just one thing, in plain text: the SHA of the commit they point to.

HEAD, in the root of .git, is the only way git knows which branch or commit you're currently "on", and it too is a plaintext file which just specifies which ref (or commit) it should go to.

## The Complete Git Data Model



now we have our complete Git data model.  
there is nothing more to the back-end than this

## Branching Workflows

```
.git
├── COMMIT_EDITMSG
└── HEAD
    └── config
    └── description
    └── hooks
        └── ...
    └── index
    └── info
        └── exclude
    └── logs
        └── ...
    └── objects
        └── 54
            └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
        └── c7
            └── dcf9ef54b124542e958c62866d8724471d77d2
        └── cc
            └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
        └── e6
            └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
        └── info
        └── pack
    └── refs
        └── heads
            └── master
        └── new_branch
    └── tags
        └── taggy
```

← Which branch you're "on"

```
$> cat .git/HEAD
ref: refs/heads/master
```

but back to this slide for a second, since we're about to talk about git checkout.  
the main point of git checkout is to change the HEAD file to point to something else,  
and then update the working directory to match.

## Branching Workflows

```
.git
  └── COMMIT_EDITMSG
  └── HEAD
  └── config
  └── description
  └── hooks
  |   └── ...
  └── index
  └── info
  |   └── exclude
  └── logs
  |   └── ...
  └── objects
  |   └── 54
  |       └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
  |   └── c7
  |       └── dcf9ef54b124542e958c62866d8724471d77d2
  |   └── cc
  |       └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
  |   └── e6
  |       └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
  |   └── info
  |   └── pack
  └── refs
      └── heads
          └── master
          └── new_branch
      └── tags
          └── taggy
```

← Which branch you're "on"

```
$> cat .git/HEAD
ref: refs/heads/master
```

```
$> git checkout new_branch
Switched to branch 'new_branch'
$> cat .git/HEAD
ref: refs/heads/new_branch
```

so here I checkout `new_branch`, and because the current branch is now `new_branch`, if we made changes to our files and committed them, the contents of `new_branch` would update to the new commit's SHA, but `master` would not

## Branching Workflows

Two notes:

- `git checkout -b yet_another_branch`
- “checkout” is different in subversion

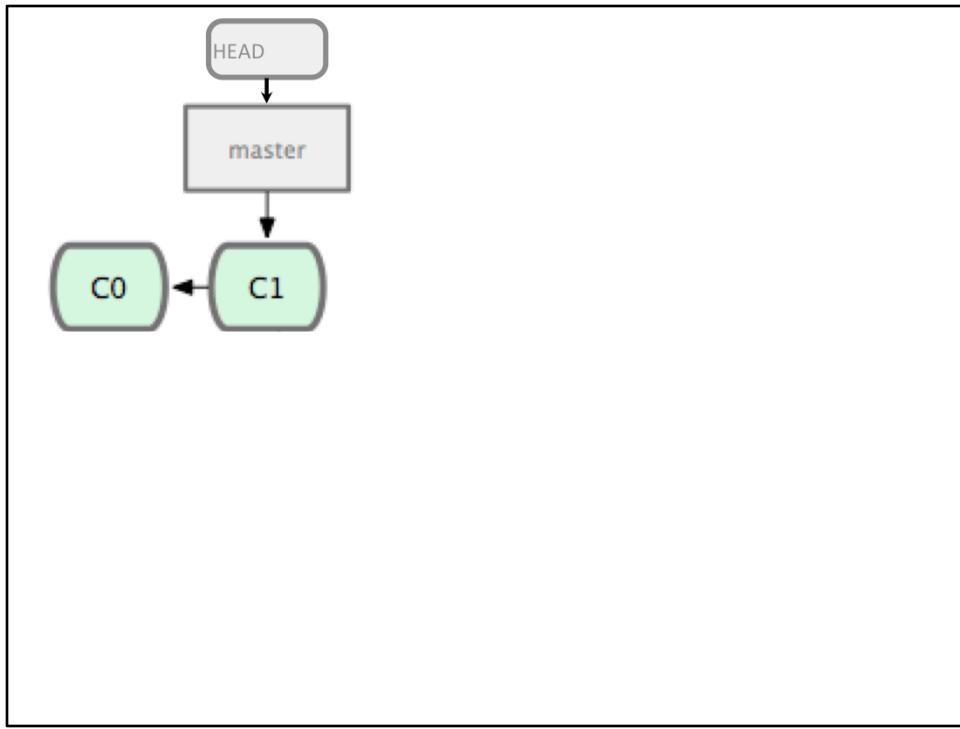
as an aside, there are many shortcuts in git, and one relevant one here is that you can create a branch and check it out in a single command with `git checkout -b` also, in case you were wondering, we’ve covered many things recently that are quite different from subversion, and “checkout” is a pretty radical one. checkout clearly does not mean the same thing in git as it does in subversion, if you’re familiar with that VCS.

## Branching Workflows

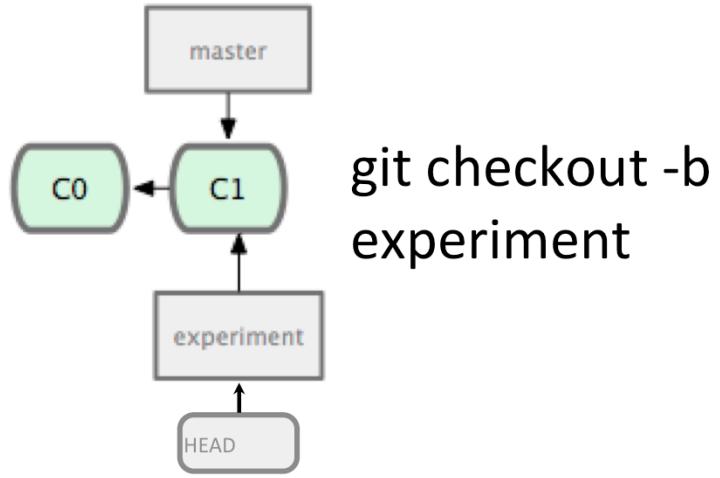
- `git branch`
- `git checkout`
- `git merge`
- `git rebase`

# Branching and Merging

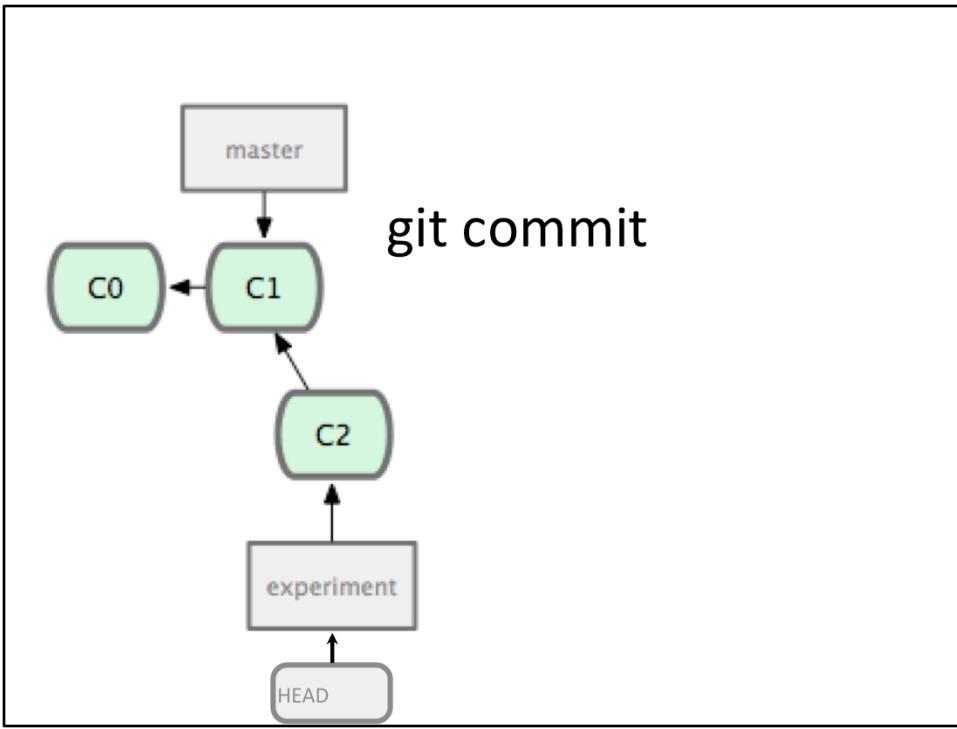
Branching and Merging



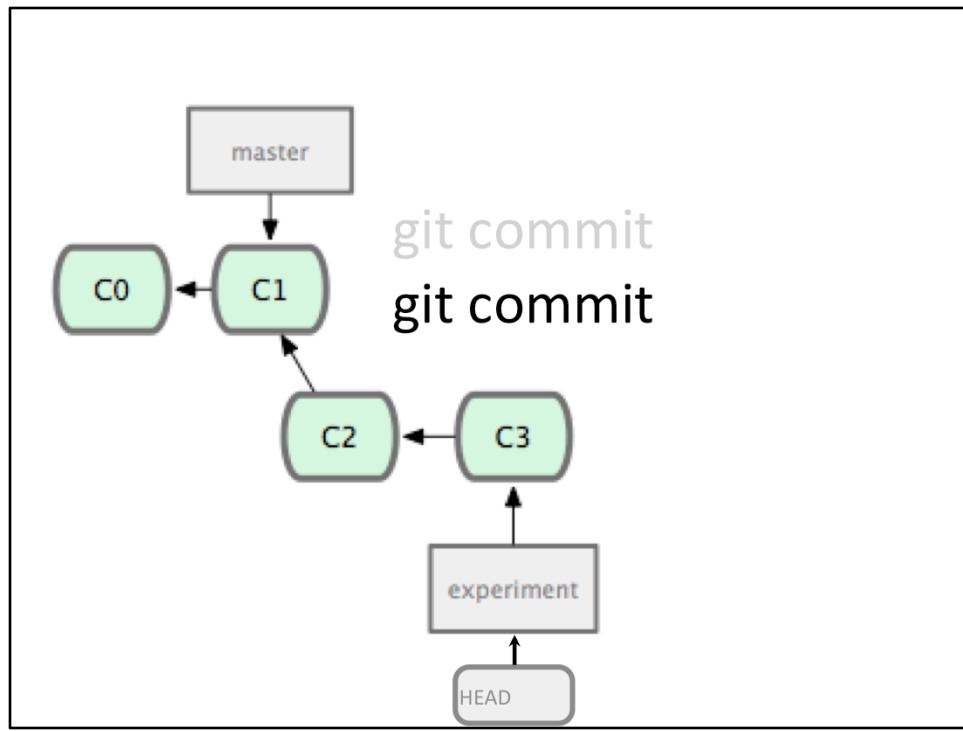
say we have a repository with two commits.  
the master branch is at C1, which is also the current branch



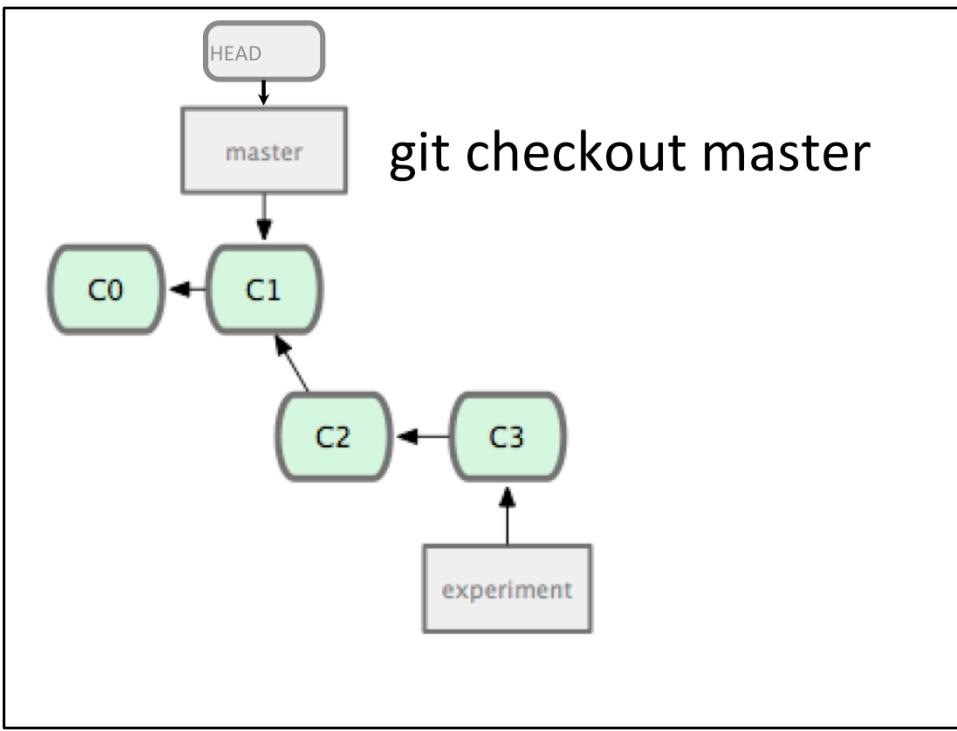
we make a new branch and check it out



and we do some work, and commit,  
and the branch we're on advances as we do

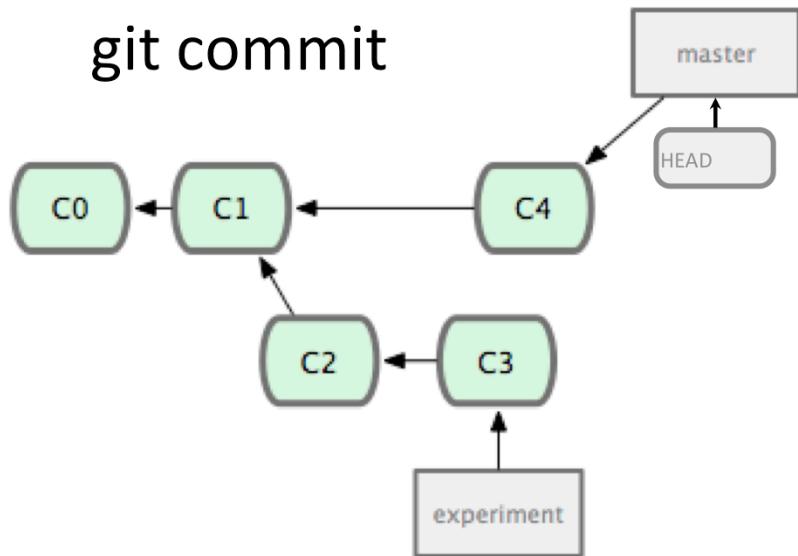


more work

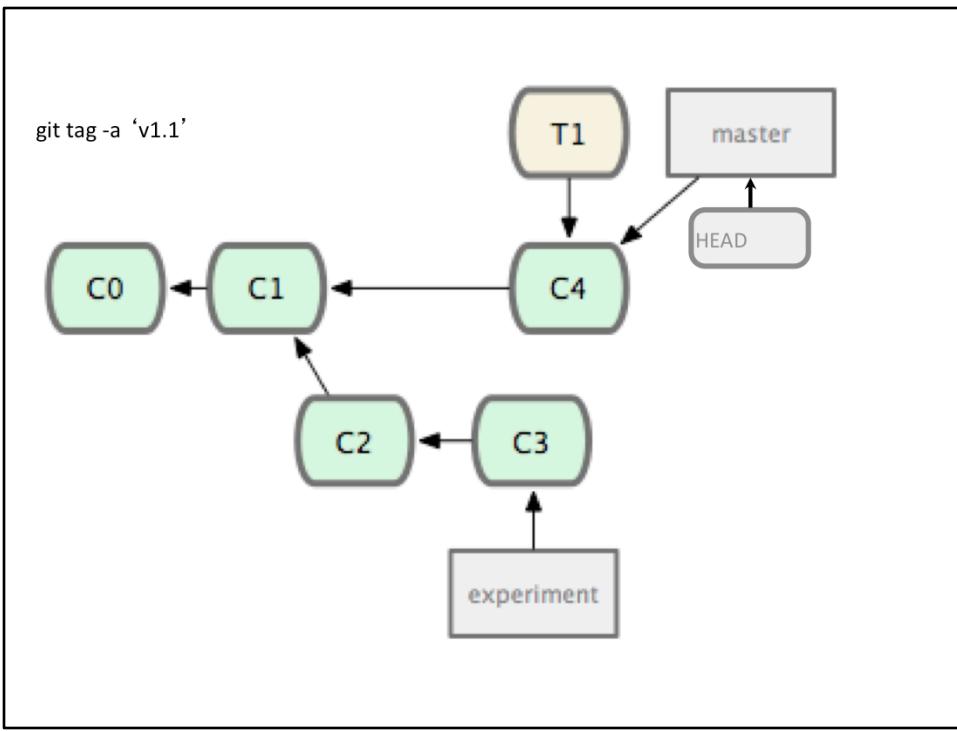


now we checkout master

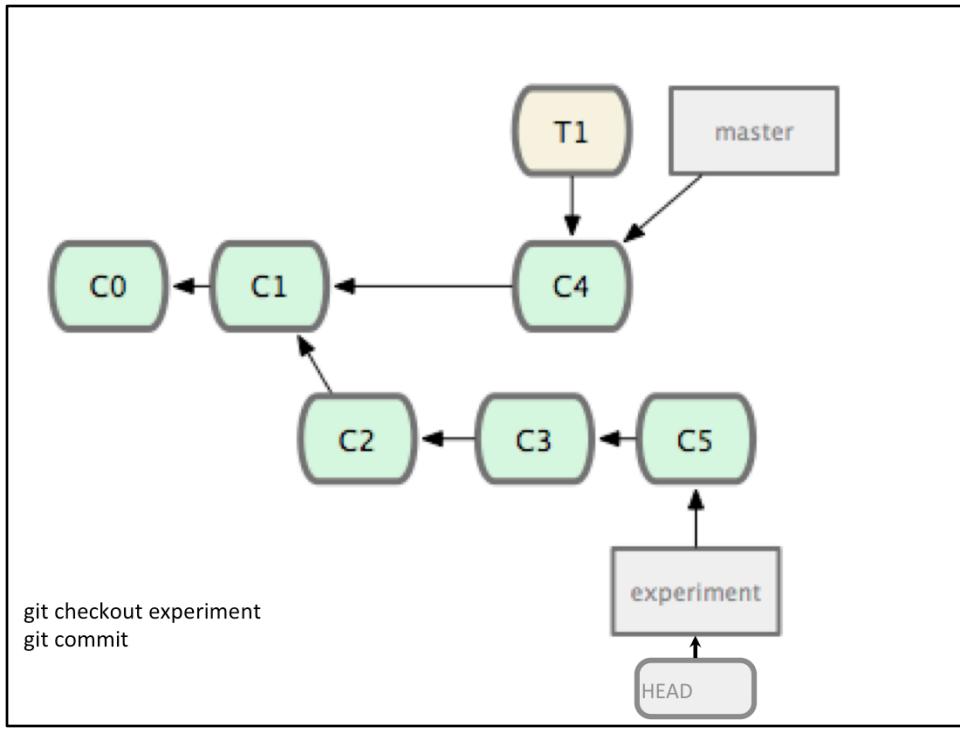
# git commit



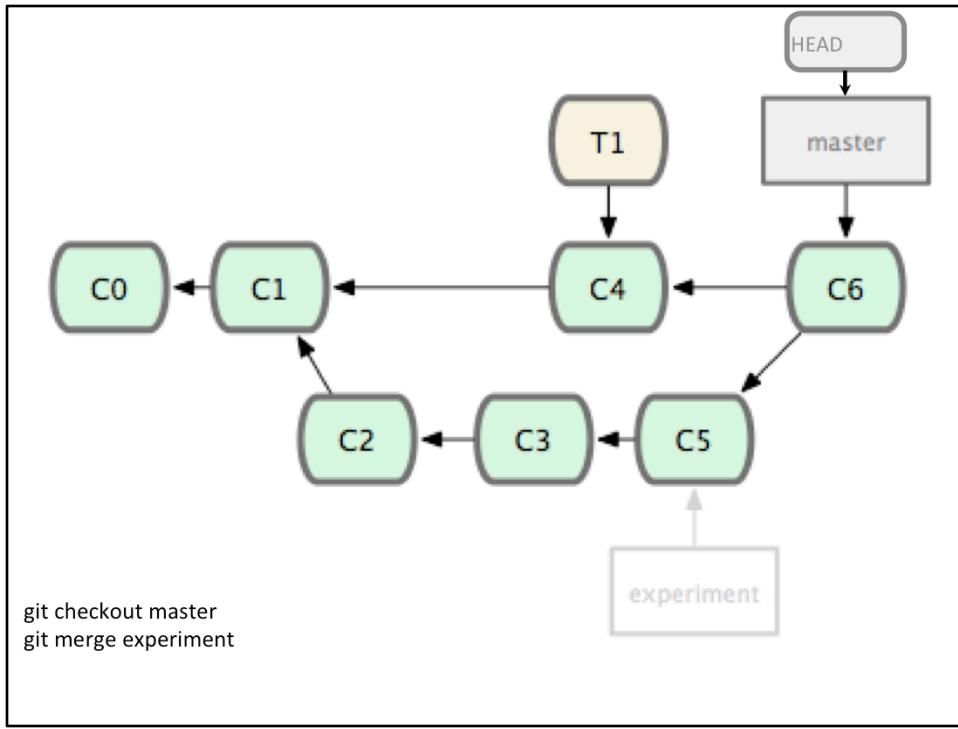
and do some work on *that* branch



and we'll add a tag



checkout experiment branch again and do some work,  
and at this point, we decide the experiment was successful, and we'd like to merge it  
back into master so people can take advantage of it



we checkout master, since that's the branch we want to merge onto,  
and we merge in the experiment branch.

all this means is that the files are merged, and a new commit is (usually) recorded to log that it was done

So... the files from the source branch tree that aren't already on the target tree are merged in, and where a file was changed in both, the conflict is resolved automatically or manually

## Merge Conflicts

```
<<<<< targetbranchSHA  
target branch changes  
foo foo foo  
=====  
source branch changes  
bar bar bar  
>>>>> sourcebranchSHA
```

if git ever has trouble merging files, usually because two branches made different changes to the same place in the same file,  
it'll put these markers in the file, and you just have to

## Merge Conflicts

```
<<<<< targetbranchSHA  
target branch changes  
foo foo foo  
-----  
source branch changes  
bar bar bar  
=>>>> sourcebranchSHA
```

manually edit the file the way you want it and delete the markers

## Merge Conflicts

foo foo foo

bar bar bar

\$> git add whatever\_file  
\$> git commit

and then save,

git add them to mark them resolved,

and commit them, and all will be well.

the messages git gives you when there are conflicts are very helpful

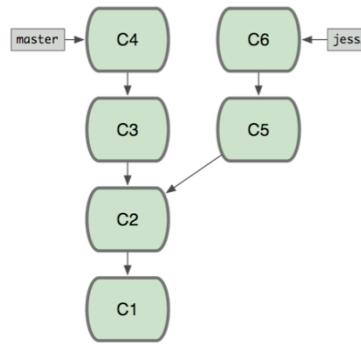
## Branching Workflows

- `git branch`
- `git checkout`
- `git merge`
- `git rebase`

# Rebasing

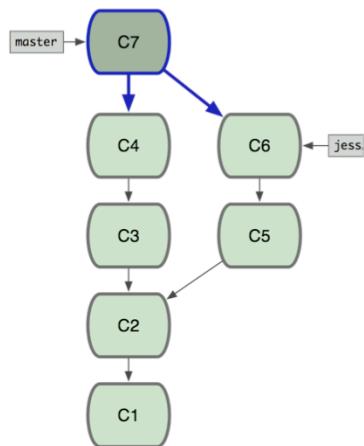
a basic use of rebase can be thought of as an alternative to merging, which results in the same files but a cleaner history in the log

## git merge



say we have this history, where we made two commits, made a new branch `jess`, made two more commits, checked out `master`, and made two more commits there as well.

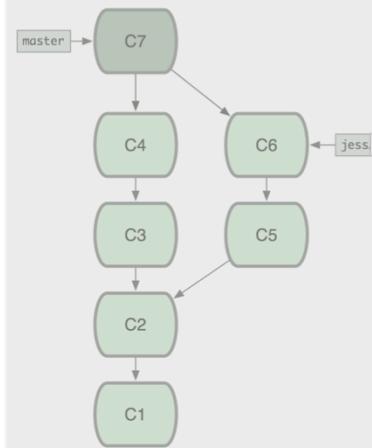
## git merge



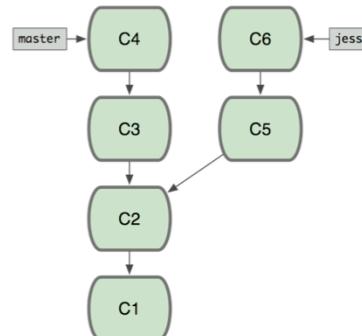
```
$> git merge jess
```

a merge at this point would create a new commit on master with two parents.

## git merge

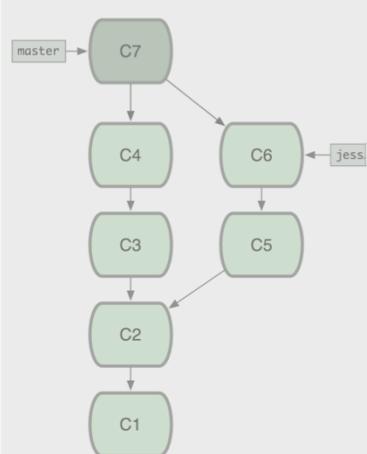


## git rebase

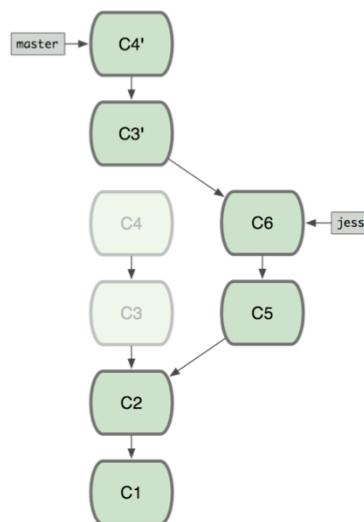


here's what would happen to the same tree if we rebased instead of merging.  
remember that master is the current branch.

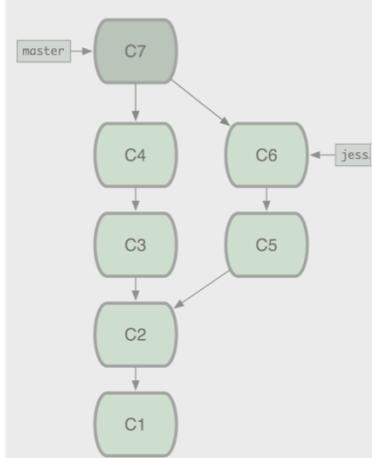
## git merge



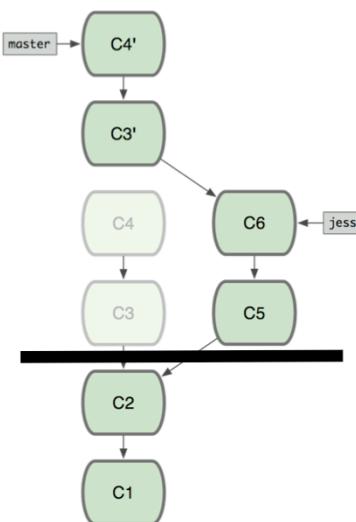
## git rebase



git merge

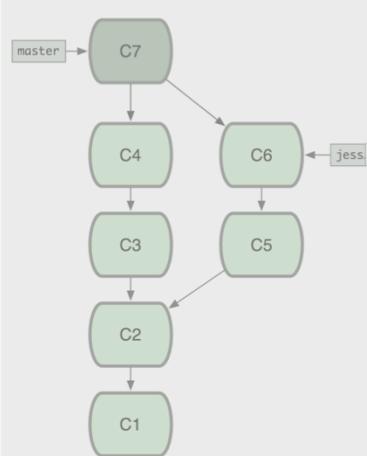


git rebase

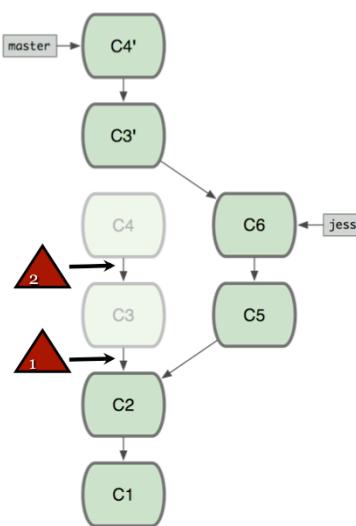


it takes all the commits since the two branches diverged

## git merge

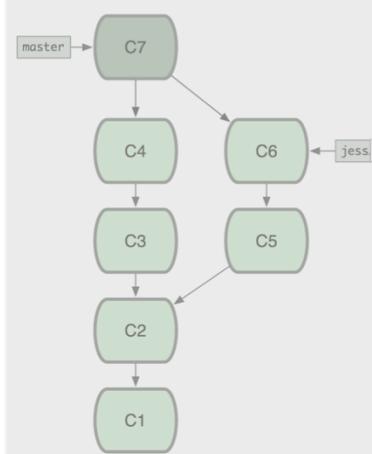


## git rebase

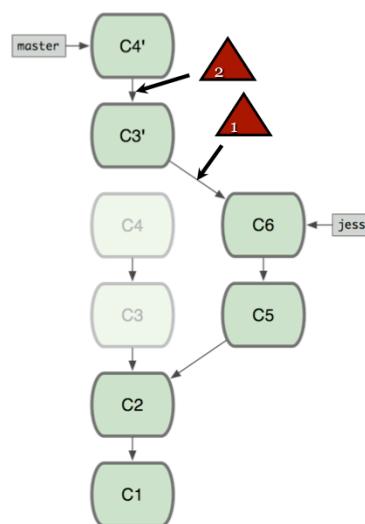


makes each commit into a patchfile

## git merge

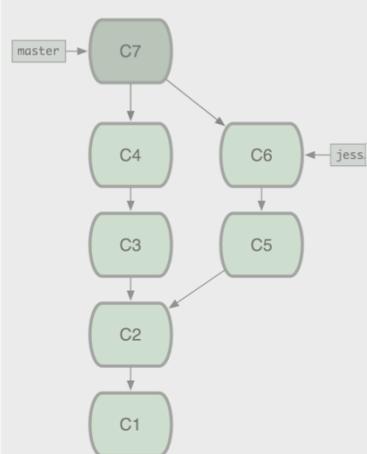


## git rebase

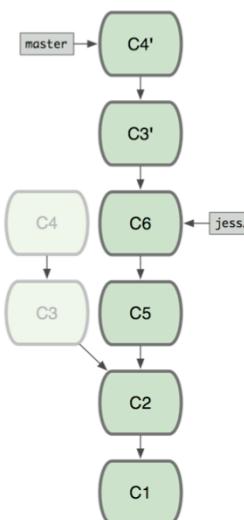


and applies them to the other branch,  
creating new commits

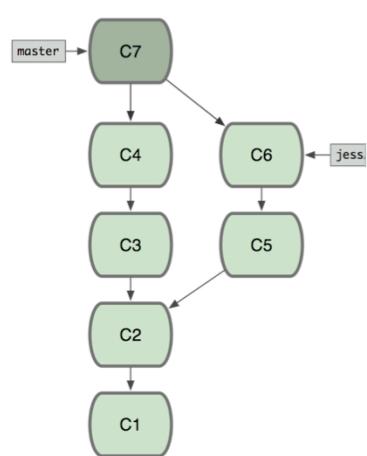
## git merge



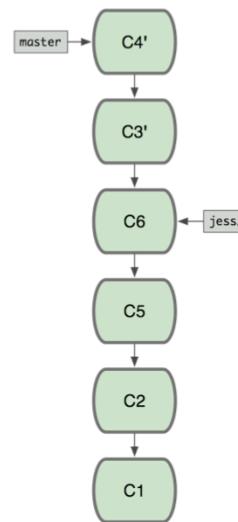
## git rebase



## git merge



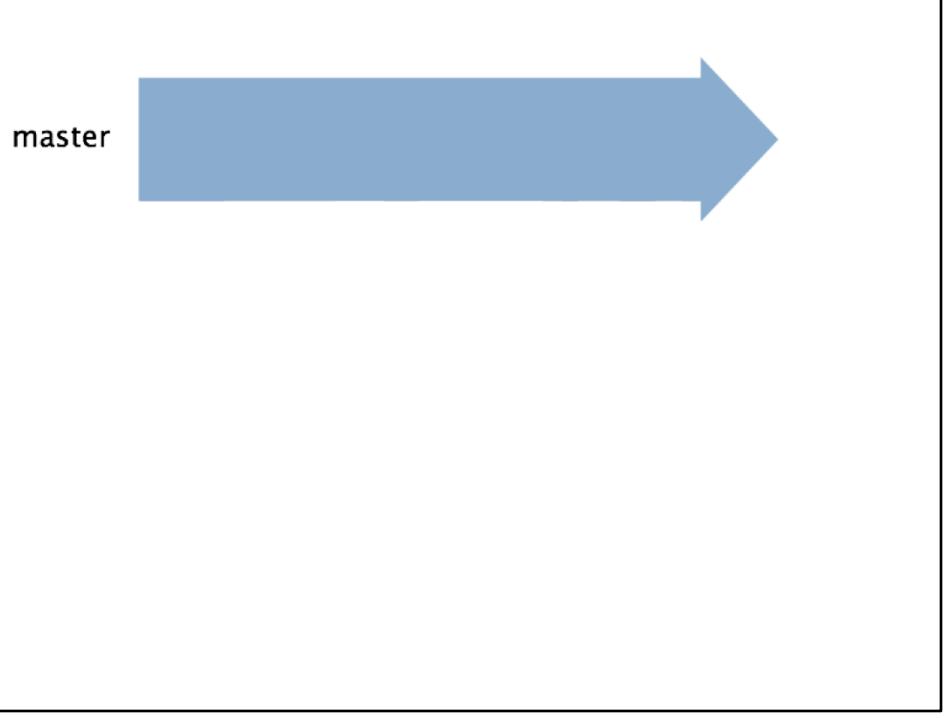
## git rebase



so you end up with a linear history that's easier to follow in the than a branched history. the “true” history can be examined using the reflog, which we'll discuss a bit more later.

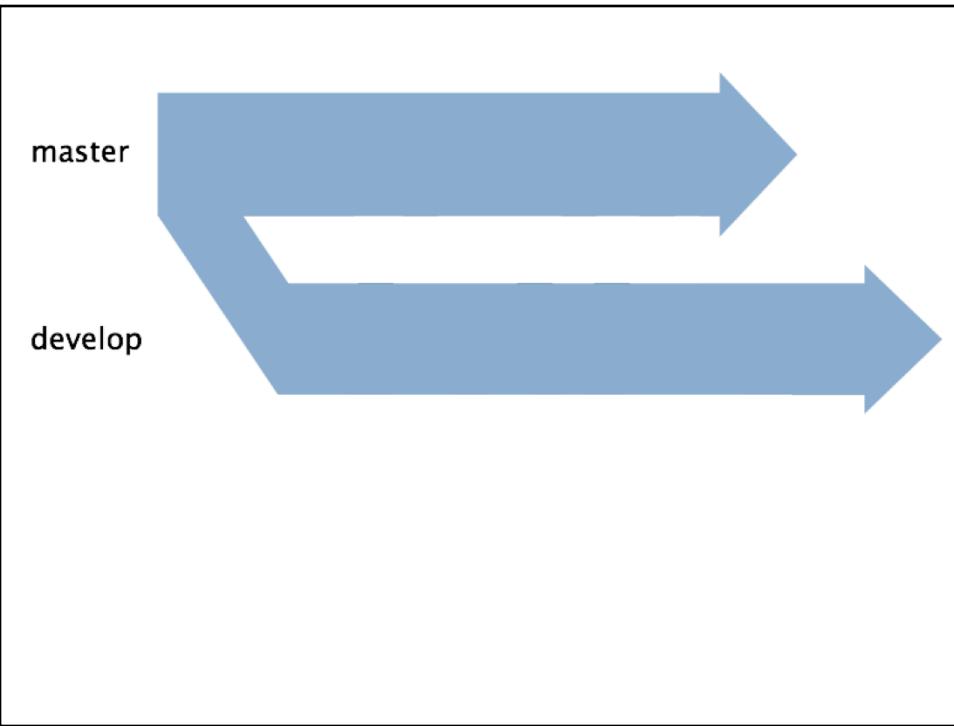
rebasing rewrites the history, so don't rebase if other people have used your commits. for instance, do not rebase commits that you've already pushed to a remote repository!

but again, we're getting ahead of ourselves again...

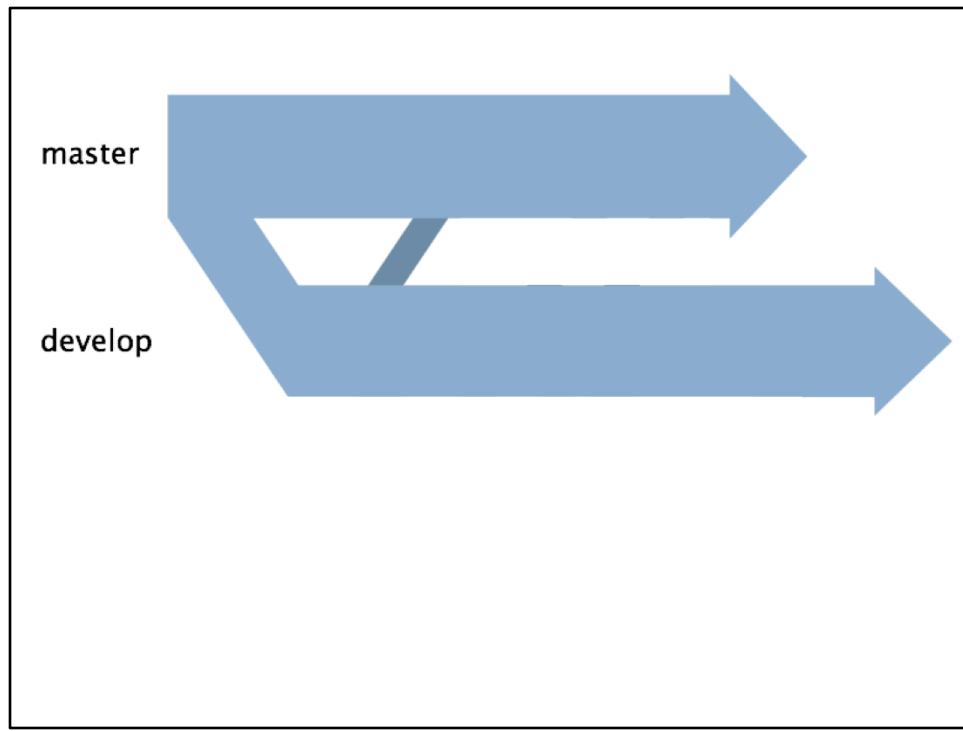


master

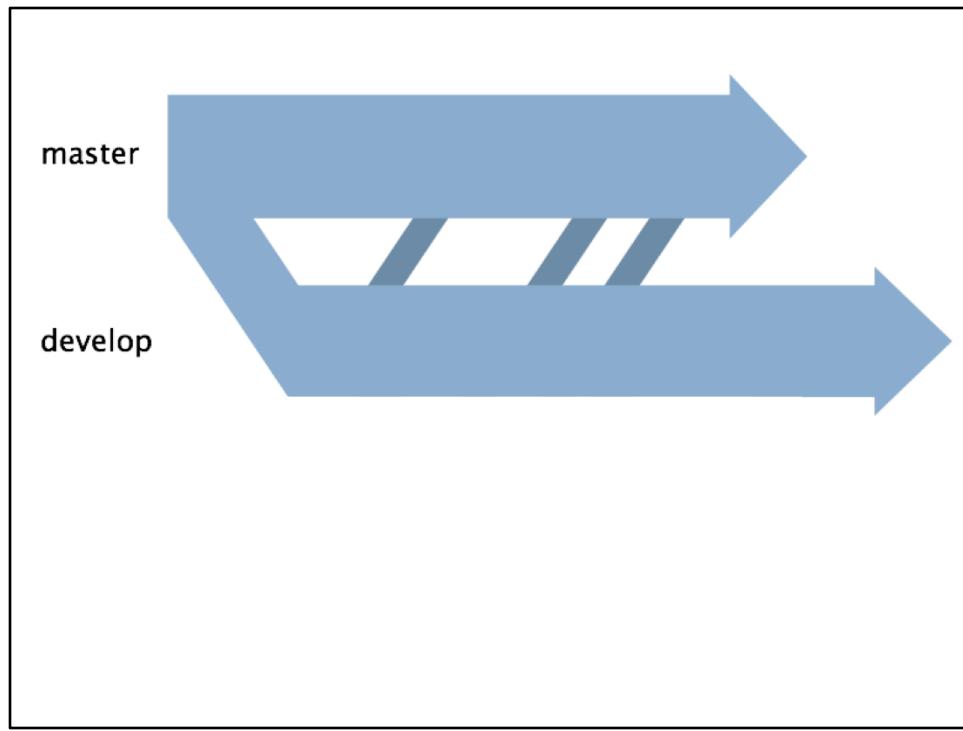
first, now that you know *how* to branch, I'm going to tell you how many successful developers branch  
in many open source projects, you'll almost always see a stable master branch,



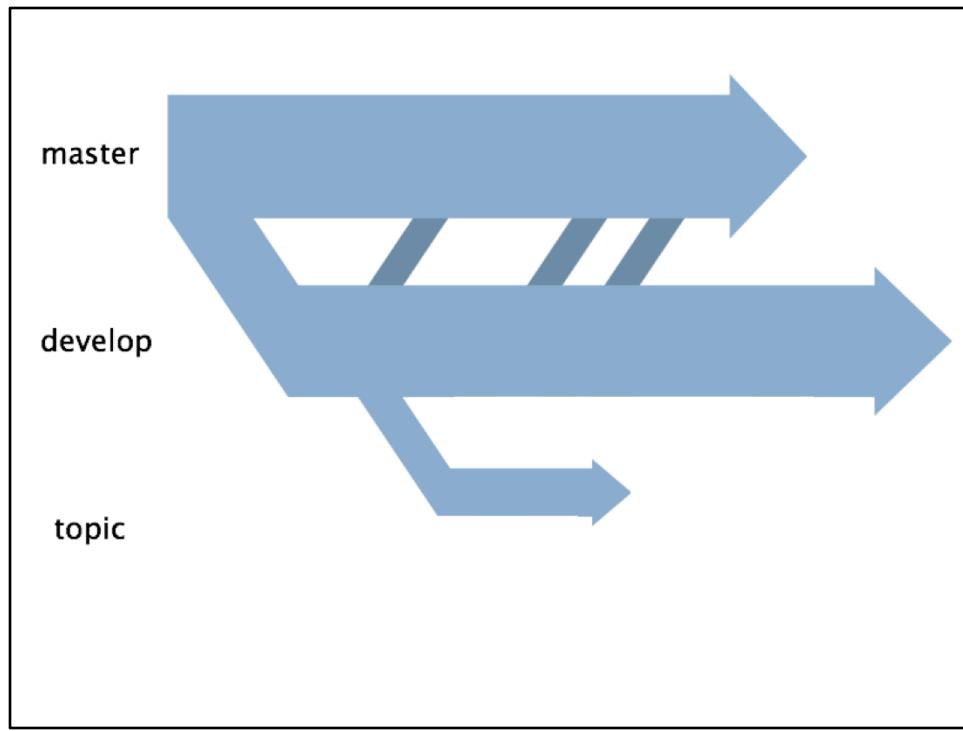
one or more development branches



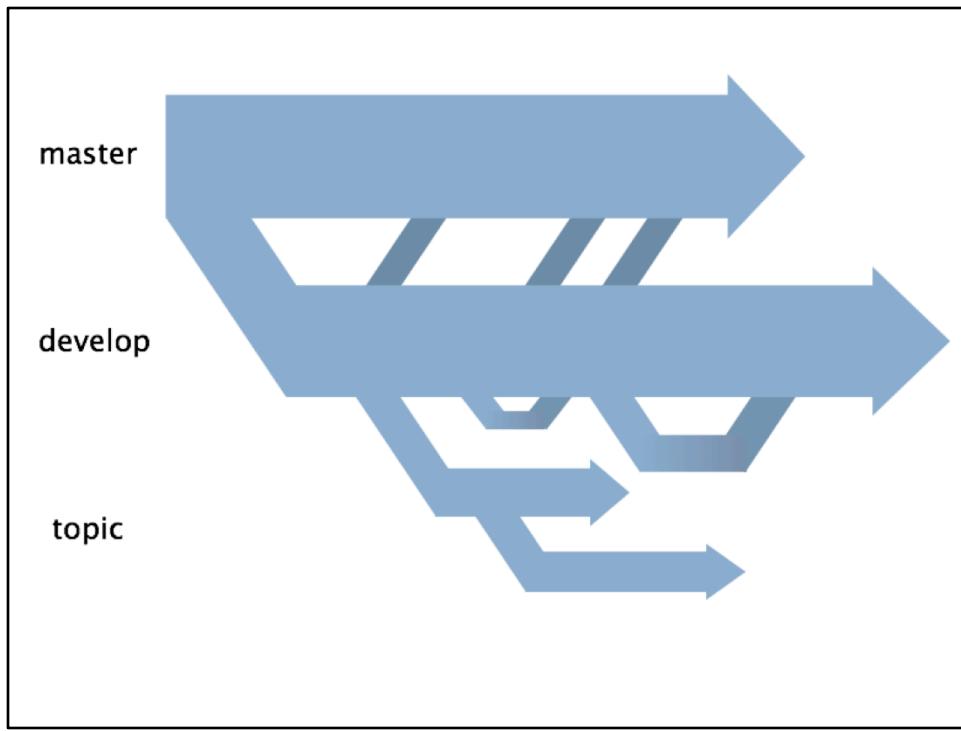
which regularly



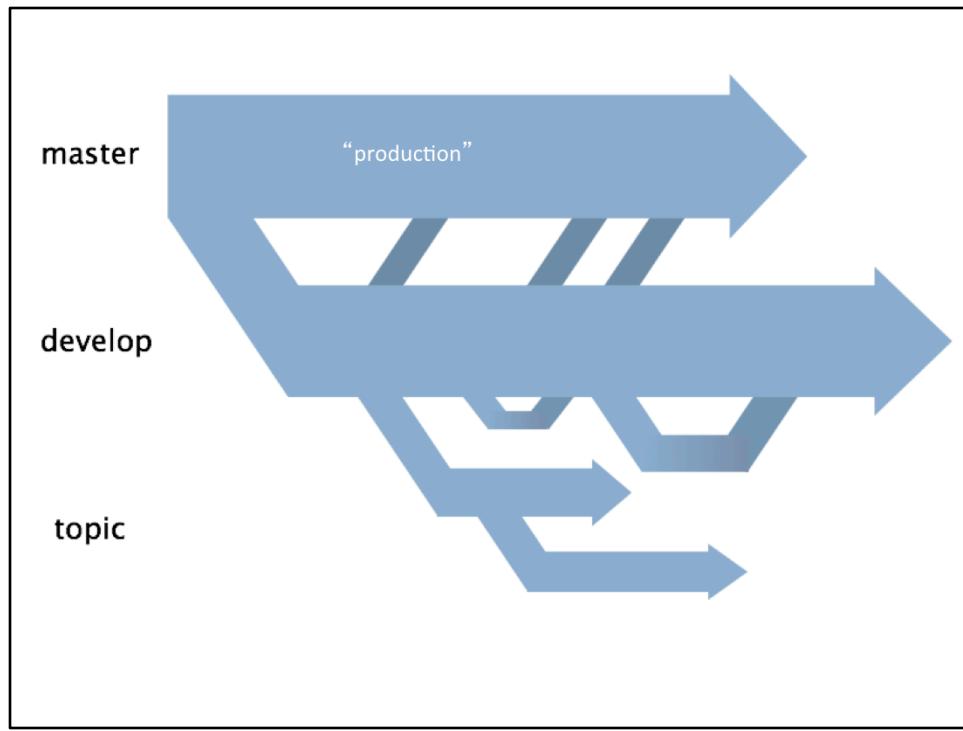
get merged back into master when they're stable



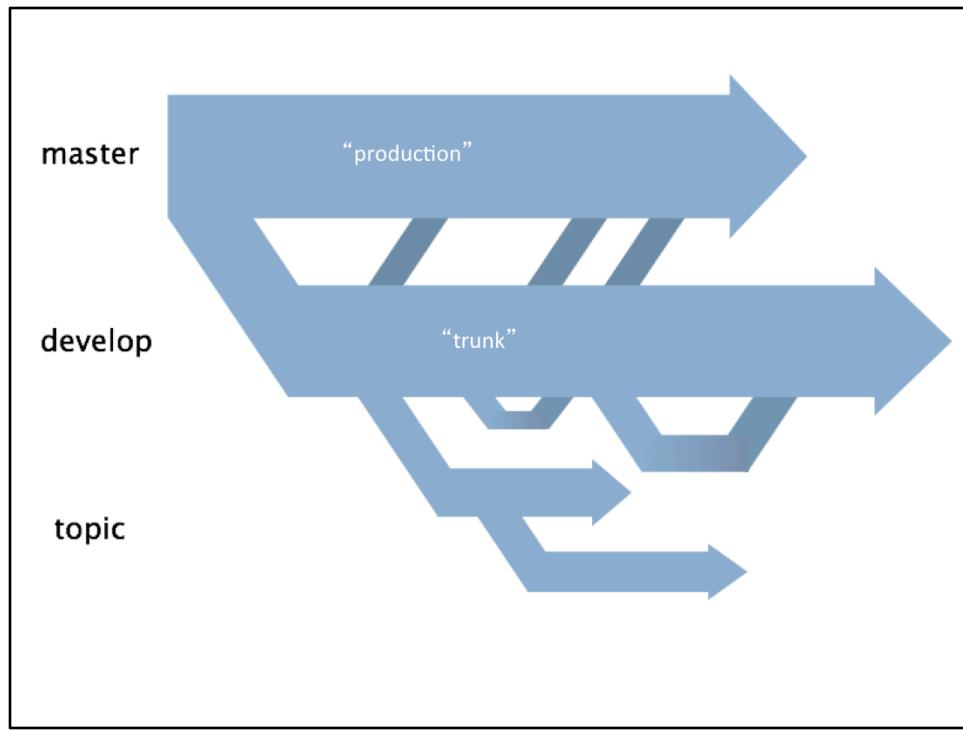
and many topic branches



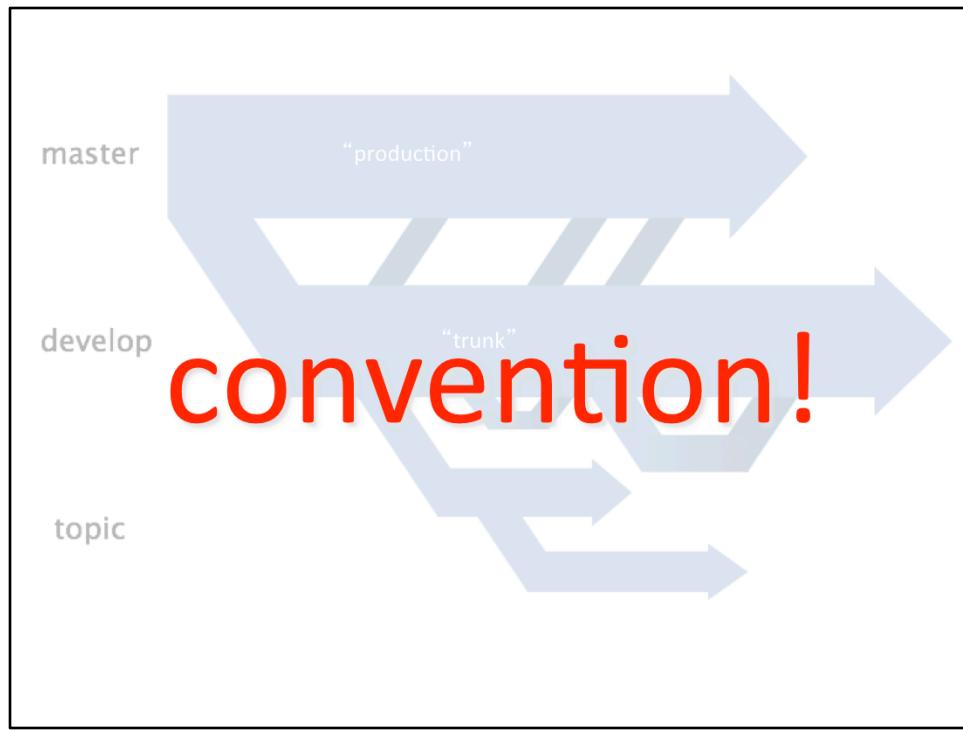
which is where most of the individual developer work is done,  
and which get merged back into development when they're complete



master usually represents the production branch,



and development is your trunk



this is, of course, mere convention, and you don't have to follow it, but it does make your work easier to understand and discuss,  
and you will be more easily able to switch back and forth between topics if you have topic branches to reduce the coupling between unrelated tasks

# Understanding Git

- ~~The Basics~~
- ~~The Git Object Database~~
- ~~History Inspection~~
- ~~Branching Workflows~~
- Collaboration
- Advanced Stuff
- Troubleshooting

on to Collaboration, and more generally, remotes.

We need to be able to work with other people, but the whole repository is on my laptop so... what?

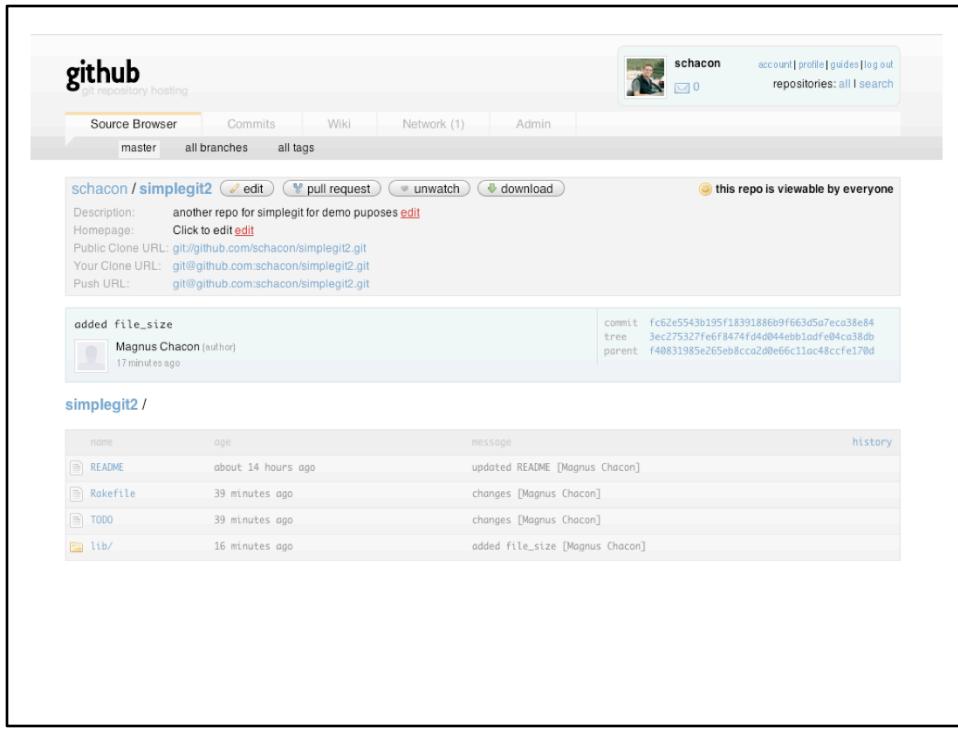
## Collaboration

- git remote
- git push
- git clone
- git fetch
- git pull

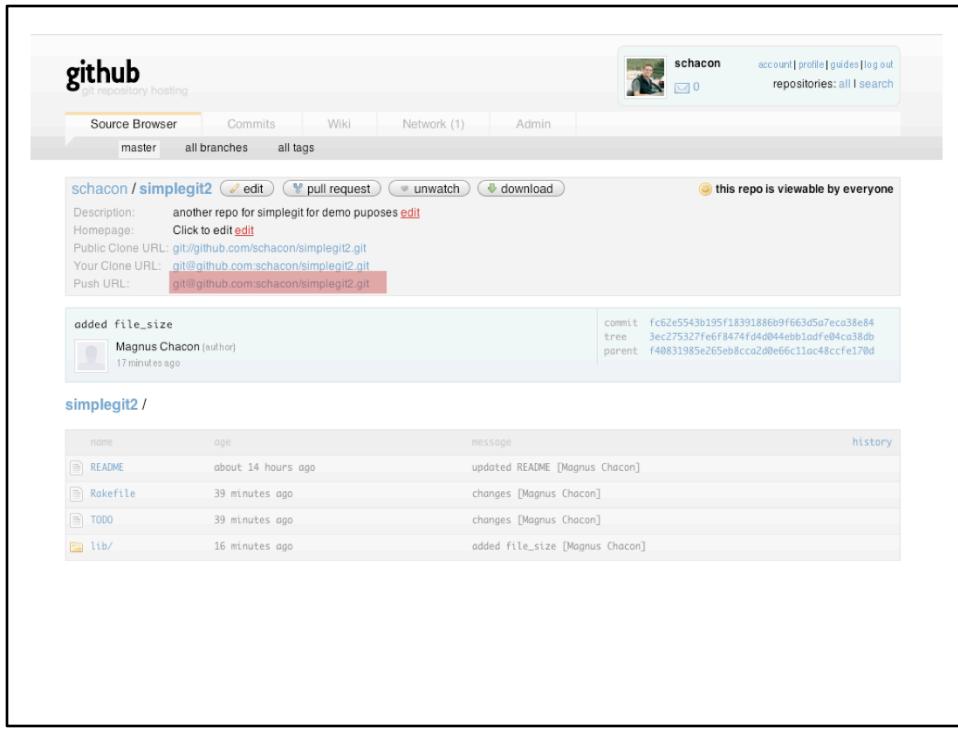
our new commands are git remote, push, clone, fetch and pull

## Collaboration

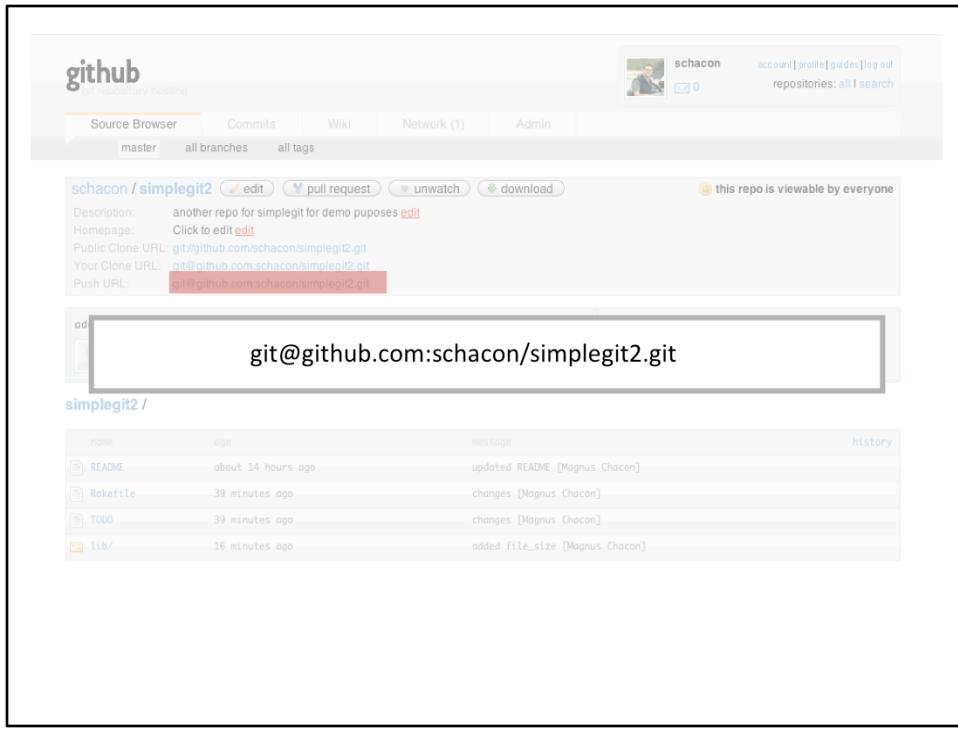
- git remote
- git push
- git clone
- git fetch
- git pull



say there's some web-accessible git repository somewhere and you already have a copy of the simplegit2 repo you've been working on, and would like to send up what you've been working on.



you find the url of the remote repository



this one uses the ssh protocol, so we can send as well as receive

```
$>git remote  
$>git remote add public git@github.com:schacon/simplegit2.git  
$>
```

we use git remote add to tell git we'd like to add a new remote repo to it's records

```
$>git remote  
$>git remote add public git@github.com:schacon/simplegit2.git  
$>
```

we'll nickname it "public"

```
$>git remote  
$>git remote add public git@github.com:schacon/simplegit2.git  
$>
```

and give it the url we found on github  
now, nothing at all happens when you add a remote, except that git has associated  
that nickname with that url

```
$>git remote
$>git remote add public git@github.com:schacon/simplegit2.git
$>git push public master
Counting objects: 115, done.
Compressing objects: 100% (73/73), done.
Writing objects: 100% (115/115), 11.20 KiB, done.
Total 115 (delta 42), reused 65 (delta 24)
refs/heads/master: 0000000000000000000000000000000000000000000000000000000000000000 -> fc6
5543b195f18391886b9f663d5a7eca38e84
To git@github.com:schacon/simplegit2.git
 * [new branch]      master -> master
$>
```

but now we can say things like `git push`

```
$>git remote
$>git remote add public git@github.com:schacon/simplegit2.git
$>git push public master
Counting objects: 115, done.
Compressing objects: 100% (73/73), done.
Writing objects: 100% (115/115), 11.20 KiB, done.
Total 115 (delta 42), reused 65 (delta 24)
refs/heads/master: 0000000000000000000000000000000000000000000000000000000000000000 -> fc6
5543b195f18391886b9f663d5a7eca38e84
To git@github.com:schacon/simplegit2.git
 * [new branch]      master -> master
$>
```

*public*

```
$>git remote
$>git remote add public git@github.com:schacon/simplegit2.git
$>git push public master
Counting objects: 115, done.
Compressing objects: 100% (73/73), done.
Writing objects: 100% (115/115), 11.20 KiB, done.
Total 115 (delta 42), reused 65 (delta 24)
refs/heads/master: 0000000000000000000000000000000000000000000000000000000000000000 -> fc6
5543b195f18391886b9f663d5a7eca38e84
To git@github.com:schacon/simplegit2.git
 * [new branch]      master -> master
$>
```

master, which, if our changes are downstream from the remote repo, will send all new commits on our master branch to the remote master branch

```
$>git remote
$>git remote add public git@github.com:schacon/simplegit2.git
$>git push public master
Counting objects: 115, done.
Compressing objects: 100% (73/73), done.
Writing objects: 100% (115/115), 11.20 KiB, done.
Total 115 (delta 42), reused 65 (delta 24)
refs/heads/master: 0000000000000000000000000000000000000000000000000000000000000000 -> fc6
5543b195f18391886b9f663d5a7eca38e84
To git@github.com:schacon/simplegit2.git
 * [new branch]      master -> master
$>git remote
public
$>
```

git remote with no arguments will tell you things about the remotes you have configured

```
$>git remote  
$>git remote add public git@github.com:schacon/simplegit2.git  
$>git push public master  
Counting objects: 115, done.  
Compressing objects: 100% (73/73), done.  
Writing objects: 100% (115/115), 11.20 KiB, done.  
Total 115 (delta 42), reused 65 (delta 24)  
refs/heads/master: 0000000000000000000000000000000000000000000000000000000000000000 -> fc6  
5543b195f18391886b9f663d5a7eca38e84  
To git@github.com:schacon/simplegit2.git  
 * [new branch]      master -> master  
$>git remote  
public  
$>git remote show public  
* remote public  
  URL: git@github.com:schacon/simplegit2.git  
  Tracked remote branches  
    master  
$>_
```

and git remote show gives you a bit more detail

## Collaboration

- `git remote`
- `git push`
- `git clone`
- `git fetch`
- `git pull`

`git clone` is exactly what it sounds like – it makes an object-for-object clone on your local machine, of a remote repository. It also makes a local branch to track the currently-active remote branch, usually `master`, and checks it out.

# Collaboration

```
$> git clone git@github.com:danielpcox/Bare-Bones-Platformer.git platformer
Cloning into 'platformer'...
remote: Counting objects: 285, done.
remote: Compressing objects: 100% (197/197), done.
remote: Total 285 (delta 138), reused 221 (delta 74)
Receiving objects: 100% (285/285), 7.46 MiB | 688 KiB/s, done.
Resolving deltas: 100% (138/138), done.
$>
```

here I am cloning a repository I have on github using the ssh protocol and url scheme,

# Collaboration

```
$> git clone git@github.com:danielpcox/Bare-Bones-Platformer.git platformer
Cloning into 'platformer'...
remote: Counting objects: 285, done.
remote: Compressing objects: 100% (197/197), done.
remote: Total 285 (delta 138), reused 221 (delta 74)
Receiving objects: 100% (285/285), 7.46 MiB | 688 KiB/s, done.
Resolving deltas: 100% (138/138), done.
$>
```

and providing an optional renaming.

# Collaboration

```
$> git clone git@github.com:danielpcox/Bare-Bones-Platformer.git platformer
Cloning into 'platformer'...
remote: Counting objects: 285, done.
remote: Compressing objects: 100% (197/197), done.
remote: Total 285 (delta 138), reused 221 (delta 74)
Receiving objects: 100% (285/285), 7.46 MiB | 688 KiB/s, done.
Resolving deltas: 100% (138/138), done.

$> ls platformer
Gemfile
Gemfile.lock
README.md
levels
lib
media
objects
platformer.rb
```

and there are the files

# Collaboration

```
$> git clone git@github.com:danielpcox/Bare-Bones-Platformer.git platformer
Cloning into 'platformer'...
remote: Counting objects: 285, done.
remote: Compressing objects: 100% (197/197), done.
remote: Total 285 (delta 138), reused 221 (delta 74)
Receiving objects: 100% (285/285), 7.46 MiB | 688 KiB/s, done.
Resolving deltas: 100% (138/138), done.

$> ls platformer
Gemfile
Gemfile.lock
README.md
levels
lib
media
objects
platformer.rb
```

as an aside, this is *another* case besides garbage-collection in which git will generate a pack file of your objects,  
it's necessary here so that an entire repository can be more efficiently transmitted

# Collaboration

```
$> git clone git@github.com:danielpcox/Bare-Bones-Platformer.git platformer
Cloning into 'platformer'...
remote: Counting objects: 285, done.
remote: Compressing objects: 100% (197/197), done.
remote: Total 285 (delta 138), reused 221 (delta 74)
Receiving objects: 100% (285/285), 7.46 MiB | 688 KiB/s, done.
Resolving deltas: 100% (138/138), done.

$> git branch
* master
```

here's that tracking branch I mentioned, which is at the same place master is, in the remote repo

## Collaboration

- `git remote`
- `git push`
- `git clone`
- `git fetch`
- `git pull`

fetch is the opposite of push

it downloads any commits you don't already have from some branch of the remote repository into a not-that-hidden branch of your repository.

No merging is done, but the commits are available for you to look at.

```
$>git remote  
github  
$>
```

say we have a repository with a remote configured

```
$>git remote
github
$>git remote show github
* remote github
  URL: git://github.com/schacon/simplegit.git
  New remote branches (next fetch will store in remotes/github)
    master
$>
```

simplegit again

```
$>git remote
github
$>git remote show github
* remote github
  URL: git://github.com/schacon/simplegit.git
  New remote branches (next fetch will store in remotes/github)
    master
$>
```

I can see that this remote is using the GIT protocol, so I can't push, but I can fetch

```
$>git remote
github
$>git remote show github
* remote github
  URL: git://github.com/schacon/simplegit.git
  New remote branches (next fetch will store in remotes/github)
    master
$>git fetch github
remote: Generating pack...
remote: Done counting 106 objects.
remote: Result has 101 objects.
remote: Deltifying 101 objects...
remote: 100% (101/101) done
remote: Total 101 (delta 35), reused 101 (delta 35)
Receiving objects: 100% (101/101), 9.49 KiB, done.
Resolving deltas: 100% (35/35), done.
From git://github.com/schacon/simplegit
 * [new branch]      master      -> github/master
$>
```

notice that we've downloaded the upstream commits from the remote master not onto our master but onto another local branch, called `github/master`

```
$>git remote
github
$>git remote show github
* remote github
  URL: git://github.com/schacon/simplegit.git
  New remote branches (next fetch will store in remotes/github)
    master
$>git fetch github
remote: Generating pack...
remote: Done counting 106 objects.
remote: Result has 101 objects.
remote: Deltifying 101 objects...
remote: 100% (101/101) done
remote: Total 101 (delta 35), reused 101 (delta 35)
Receiving objects: 100% (101/101), 9.49 KiB, done.
Resolving deltas: 100% (35/35), done.
From git://github.com/schacon/simplegit
 * [new branch]      master      -> github/master
$>git branch -a
  develop
  master
* story95
  github/master
$>
```

which you can see using `-a` on `git branch`.

now, this is an important point – remotes aren't anything new to the git data model

```
...  
└─ refs  
    ├─ heads  
    |   └─ master  
    └─ remotes  
        └─ origin  
            ├─ master  
            └─ HEAD  
        ...
```

.git/refs/remotes/nickname/branchname

they are simply branches, and appear underneath `.git/refs/remotes/nickname/branchname`

```
$>git remote
github
$>git remote show github
* remote github
  URL: git://github.com/schacon/simplegit.git
  New remote branches (next fetch will store in remotes/github)
    master
$>git fetch github
remote: Generating pack...
remote: Done counting 106 objects.
remote: Result has 101 objects.
remote: Deltifying 101 objects...
remote: 100% (101/101) done
remote: Total 101 (delta 35), reused 101 (delta 35)
Receiving objects: 100% (101/101), 9.49 KiB, done.
Resolving deltas: 100% (35/35), done.
From git://github.com/schacon/simplegit
 * [new branch]      master      -> github/master
$>git branch -a
  develop
  master
* story95
  github/master
$>_
```

now this branch can be used in every way like a normal local branch, except that it cannot be checked out.

you can merge from it, rebase onto it, cherry-pick from it, browse its logs... etc.

## Collaboration

- `git remote`
- `git push`
- `git clone`
- `git fetch`
- `git pull`

`git pull`

this one will be easy

a pull is *not* the opposite of a push.

## Collaboration

**pull = fetch + merge**

the default behavior of a pull is a fetch, followed by a merge  
since I've talked about both of those, I won't spend any more time on it

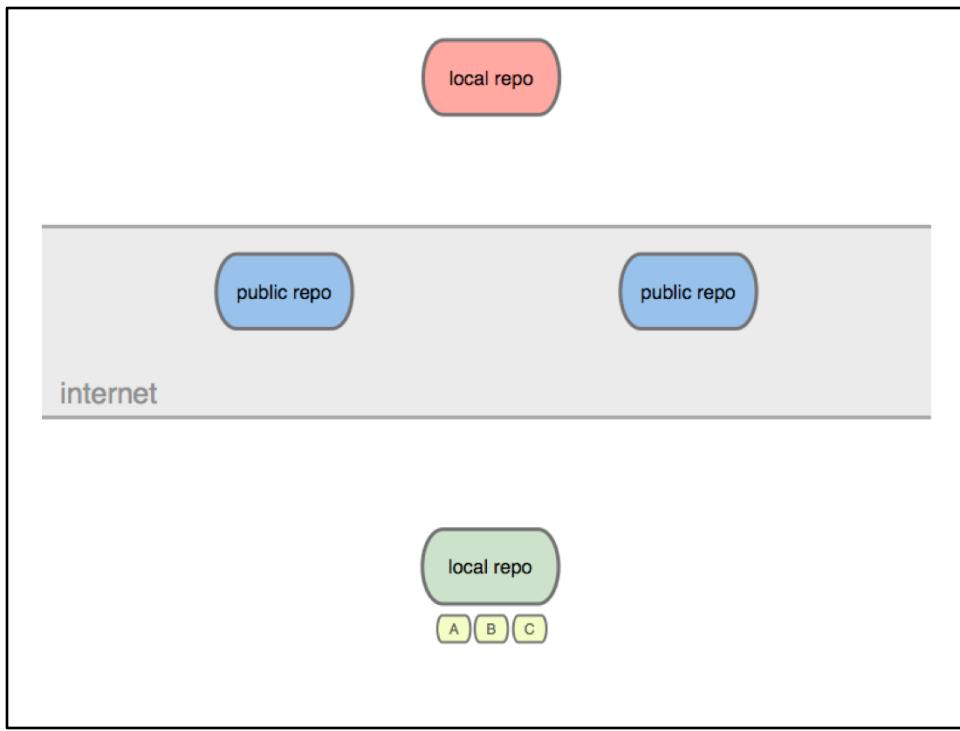
## Collaboration

- `git remote`
- `git push`
- `git clone`
- `git fetch`
- `git pull`

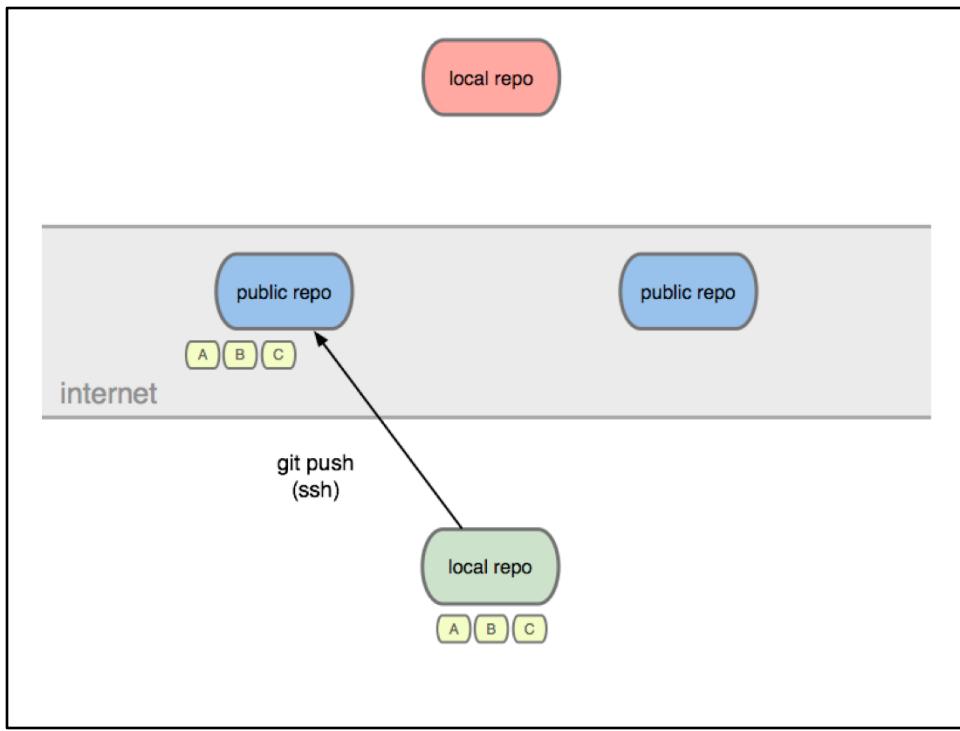
those are the commands...

## **remote workflow**

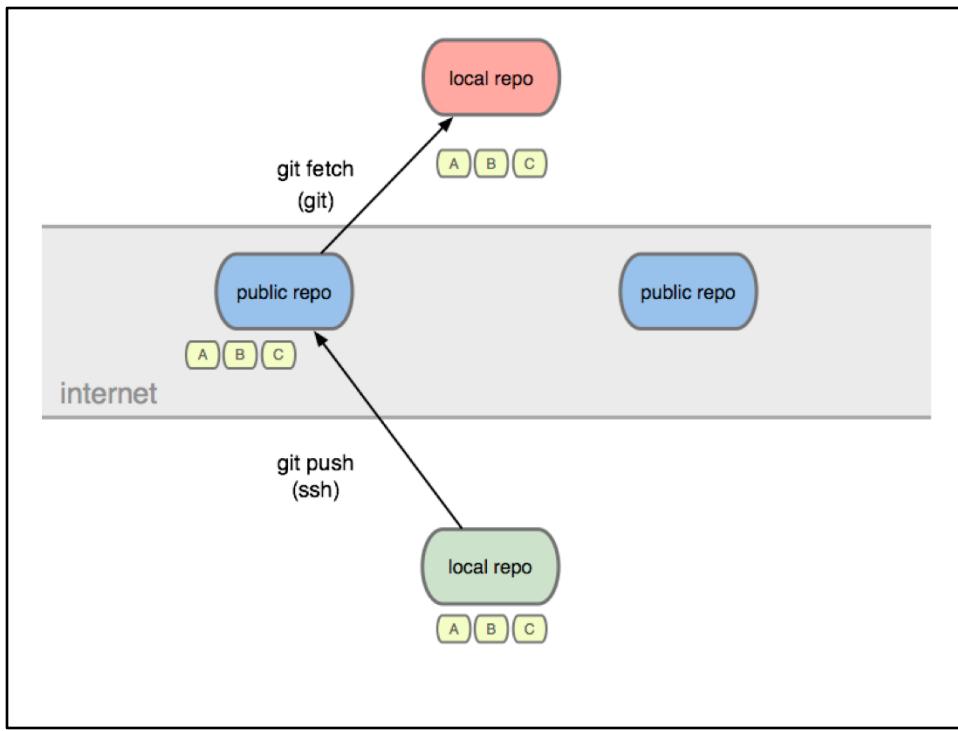
here are some practical examples



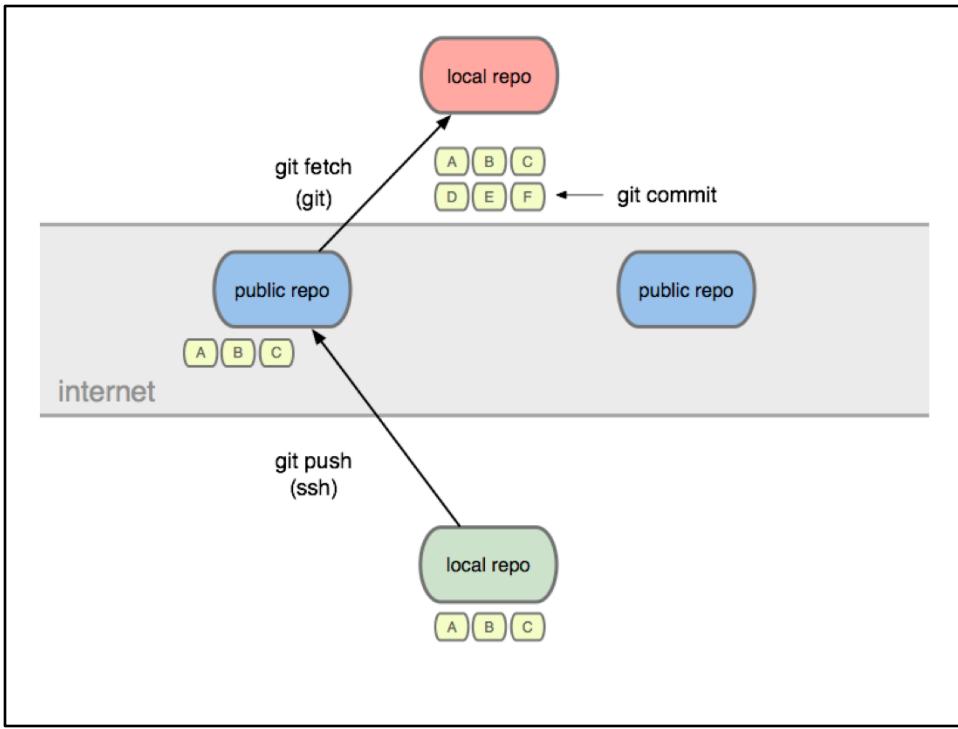
say we have this setup, and in our local repo, a bunch of commits.



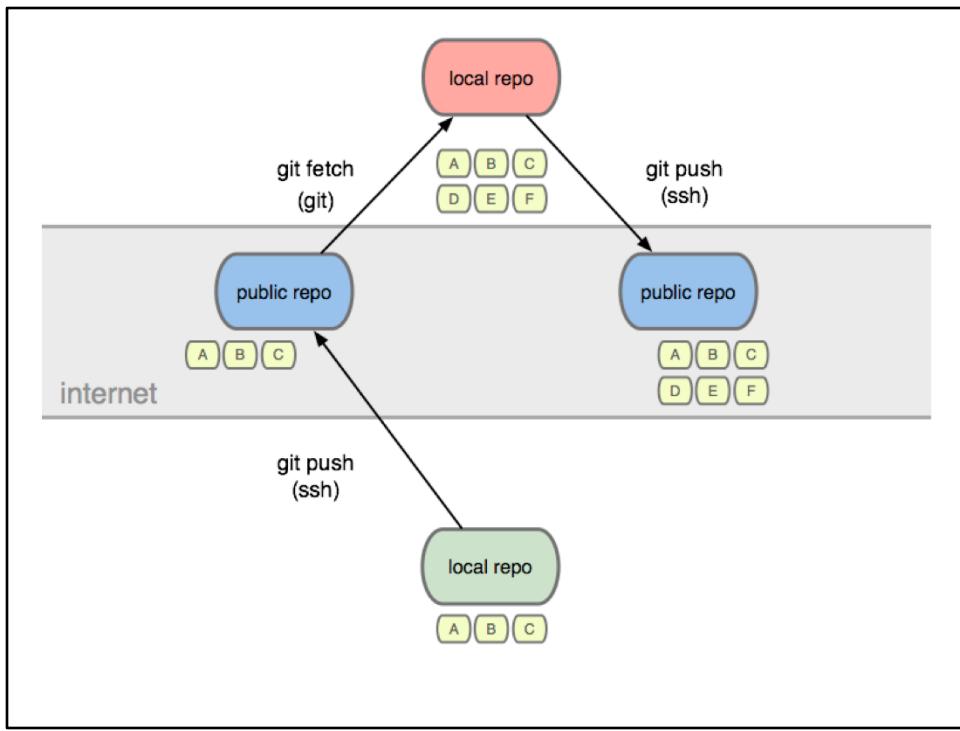
we can git push over ssh to a public repo we control



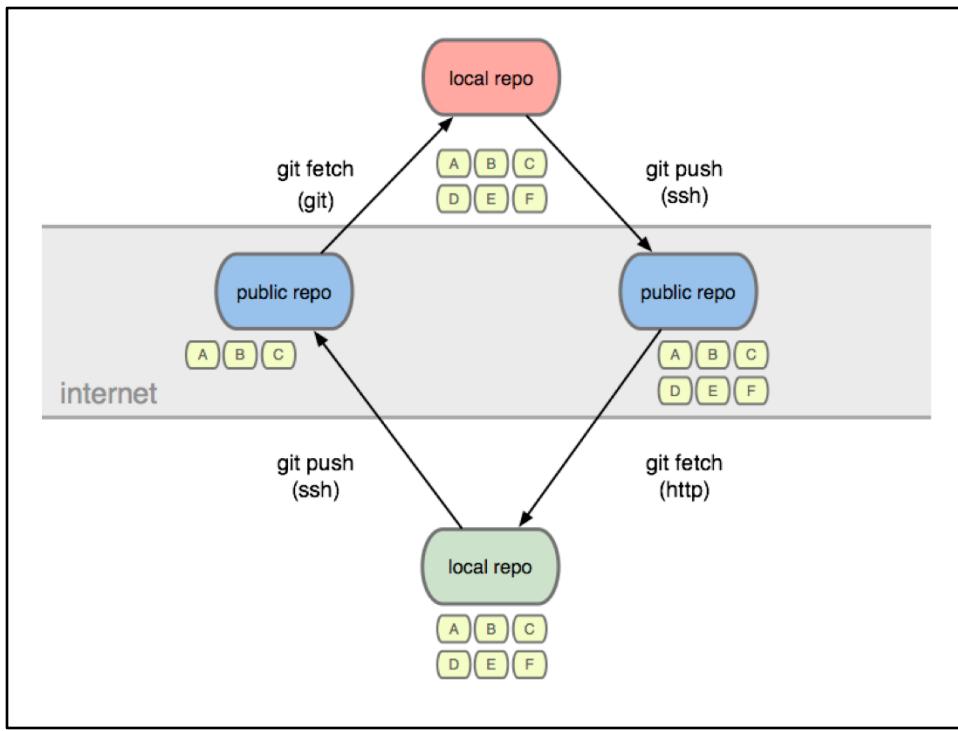
so someone else can git fetch from it to get our commits



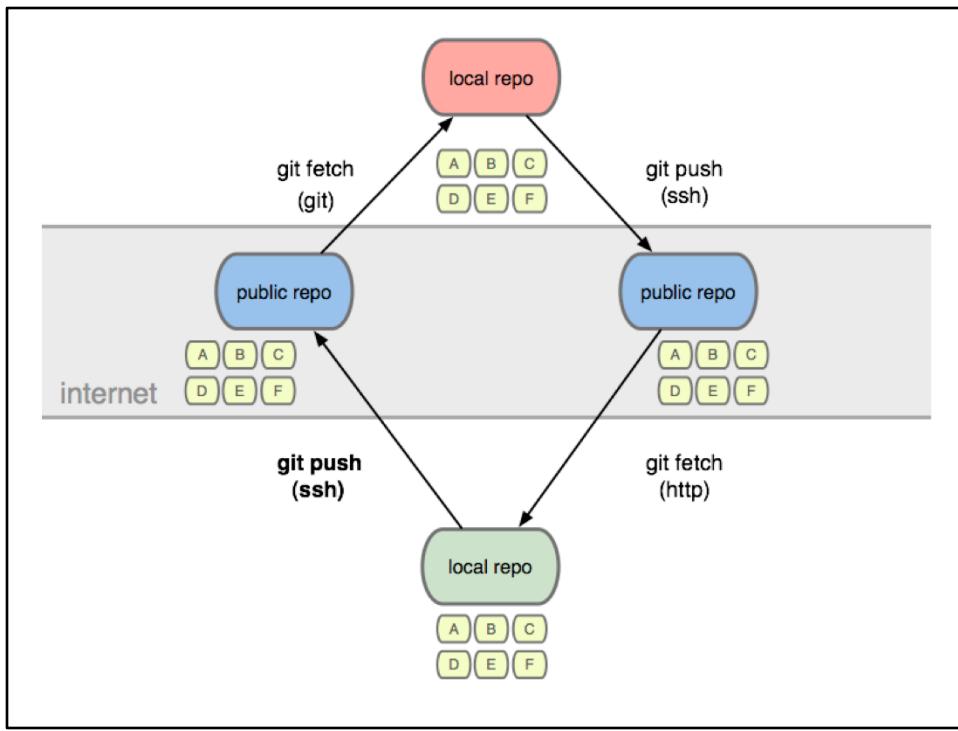
they may add their own,



and push to their own public repo

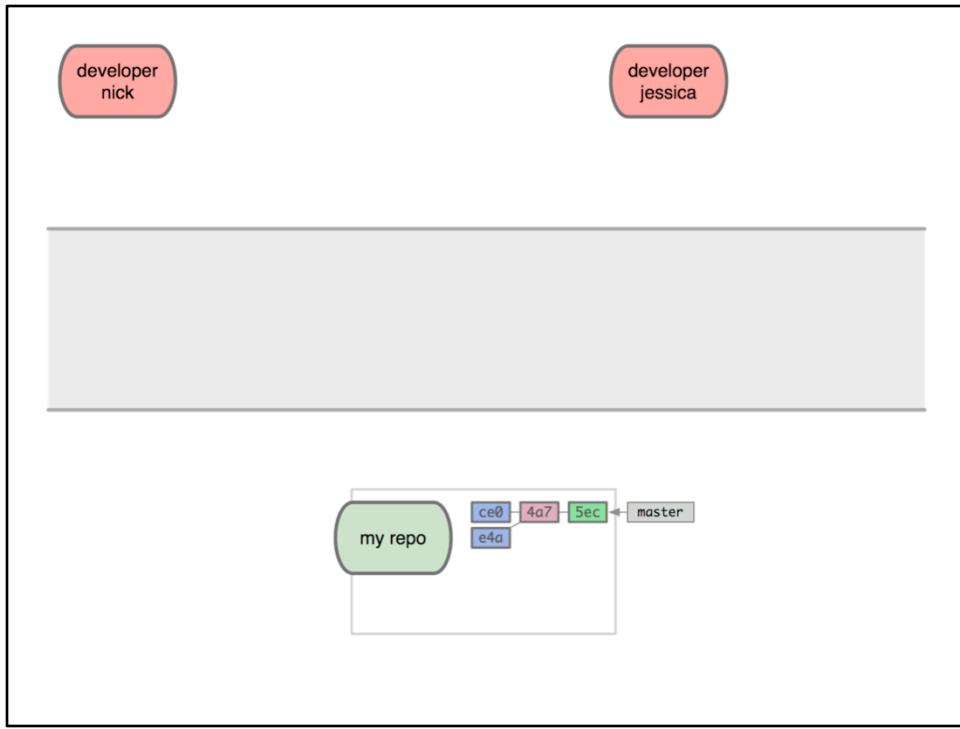


which we can fetch from to get (notice) *only* their three new commits.

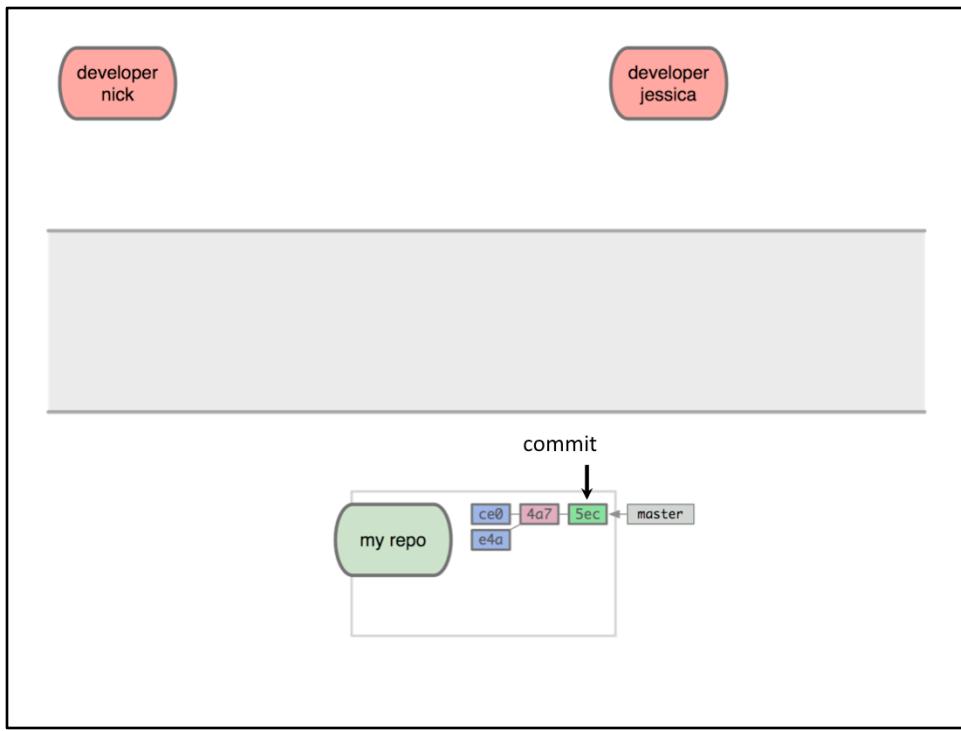


which we can send to our repo if we like

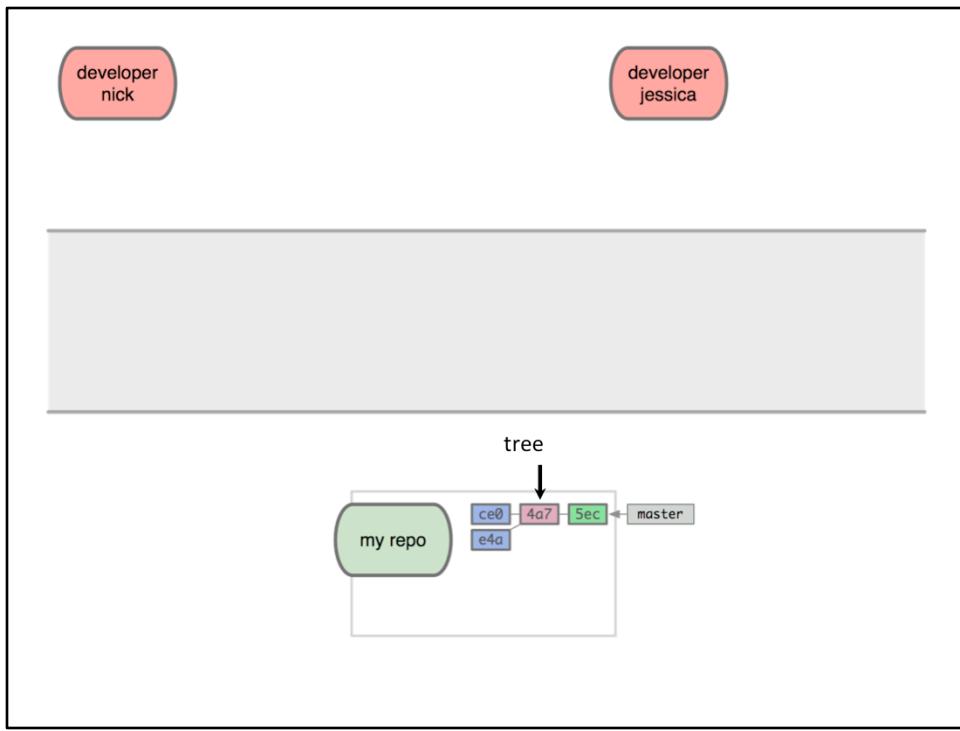
multiple remotes



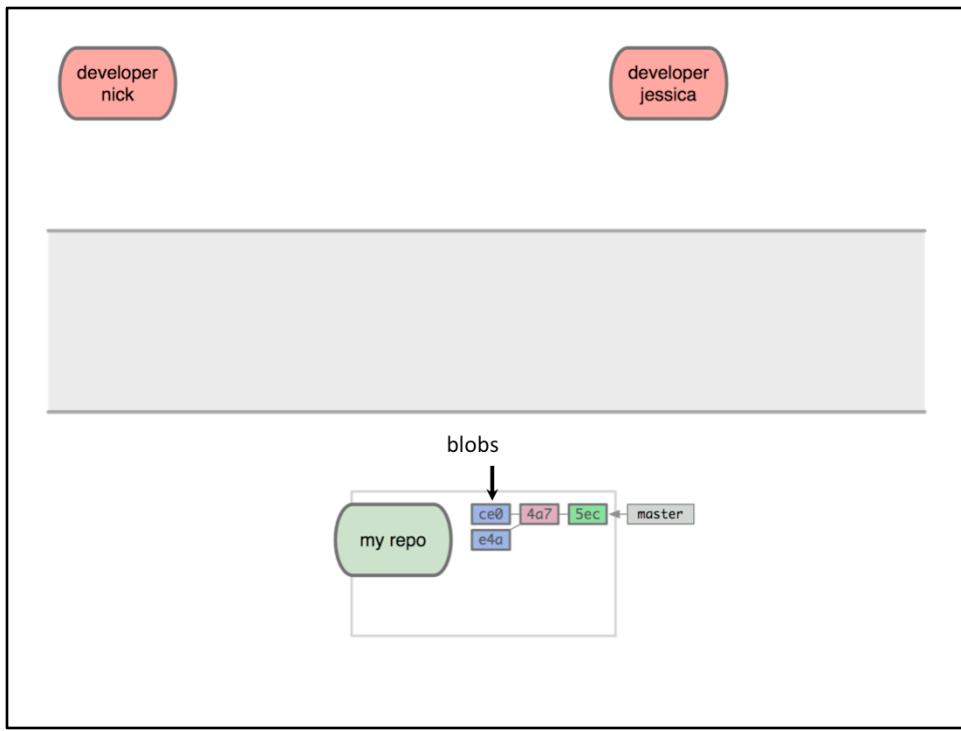
this one's a bit more complicated  
this is the github model, and is quite social



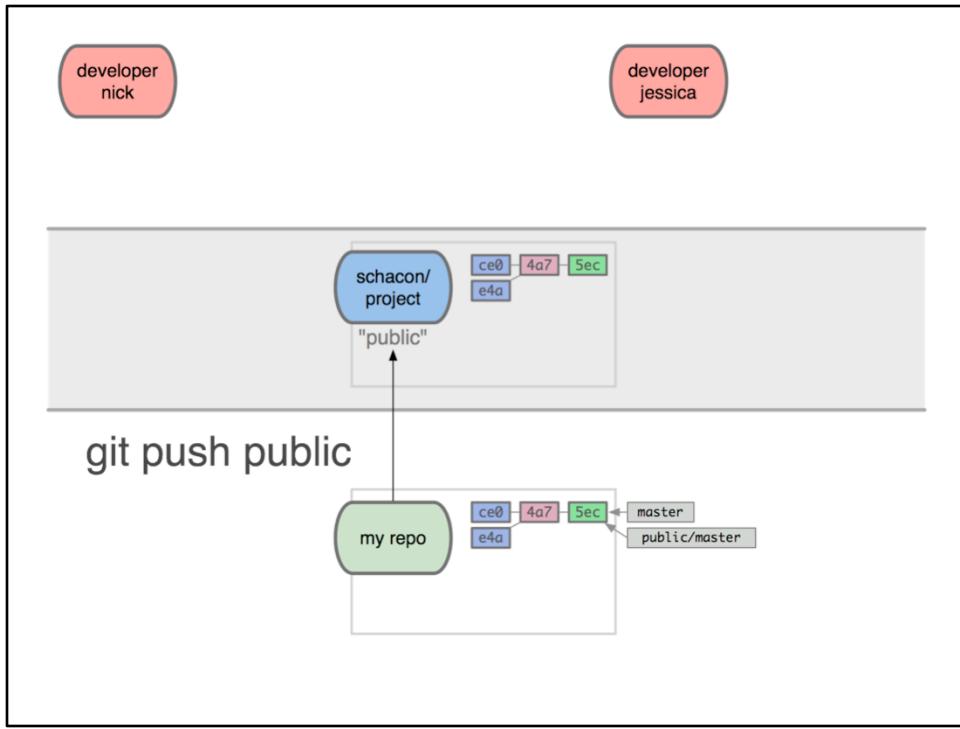
commit



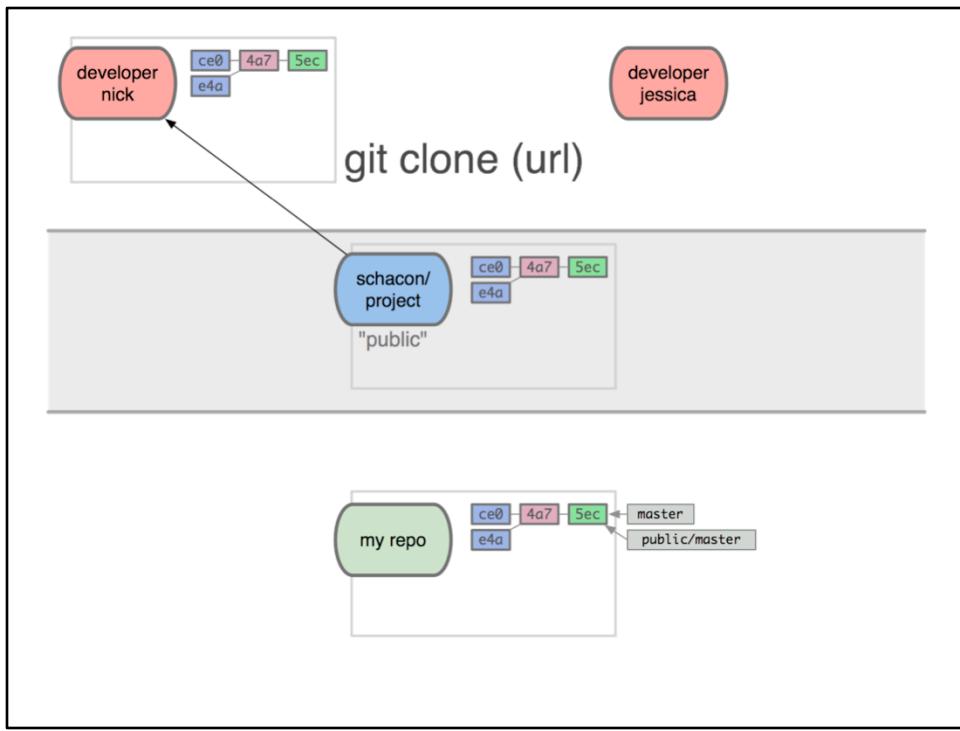
tree



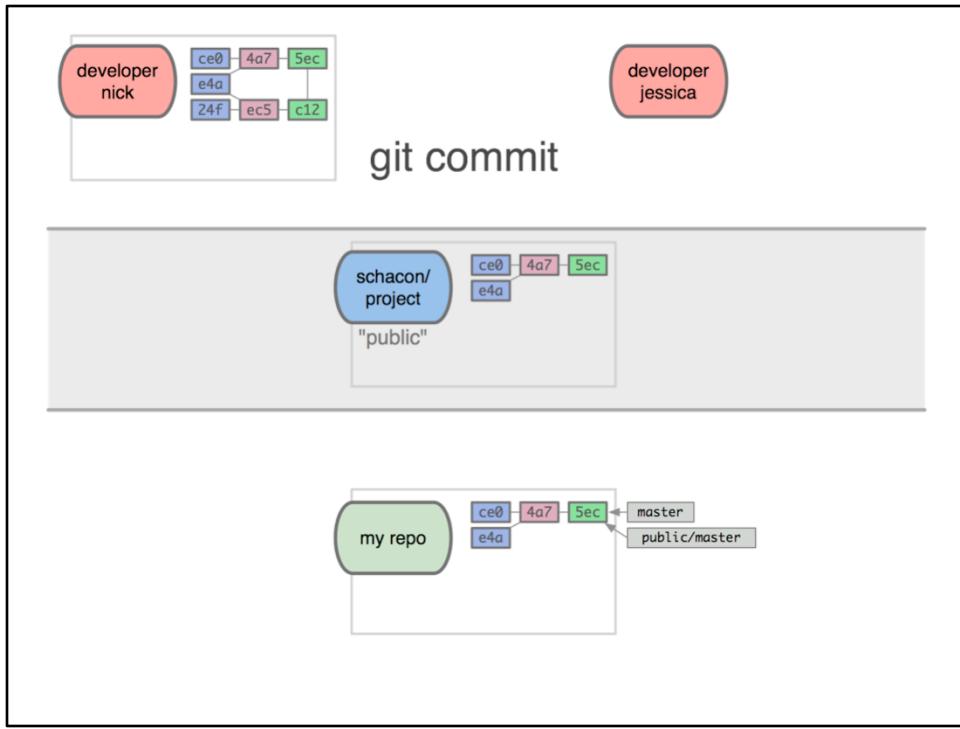
blobs



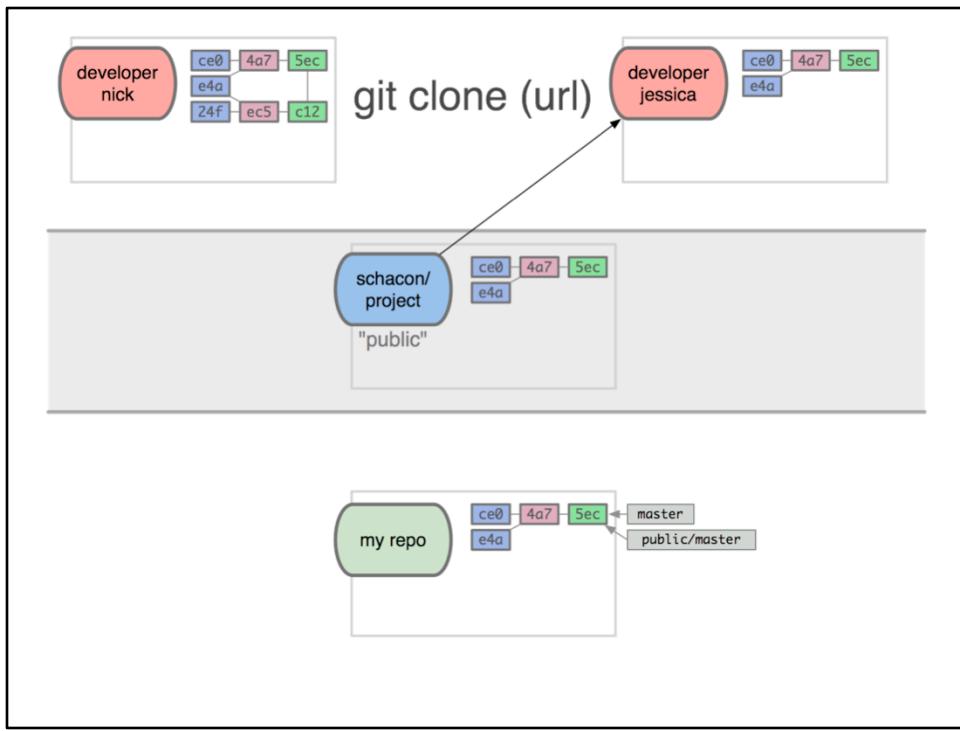
i have a public project I'm working on, and that remote is nicknamed "public" so I push my commit tree there



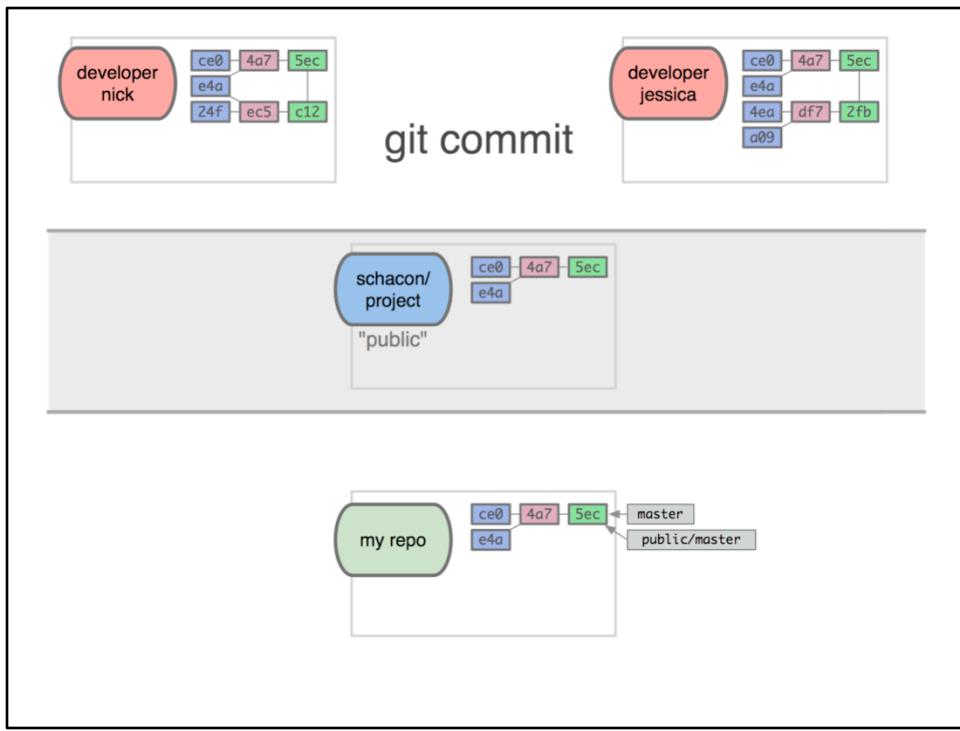
nick clones



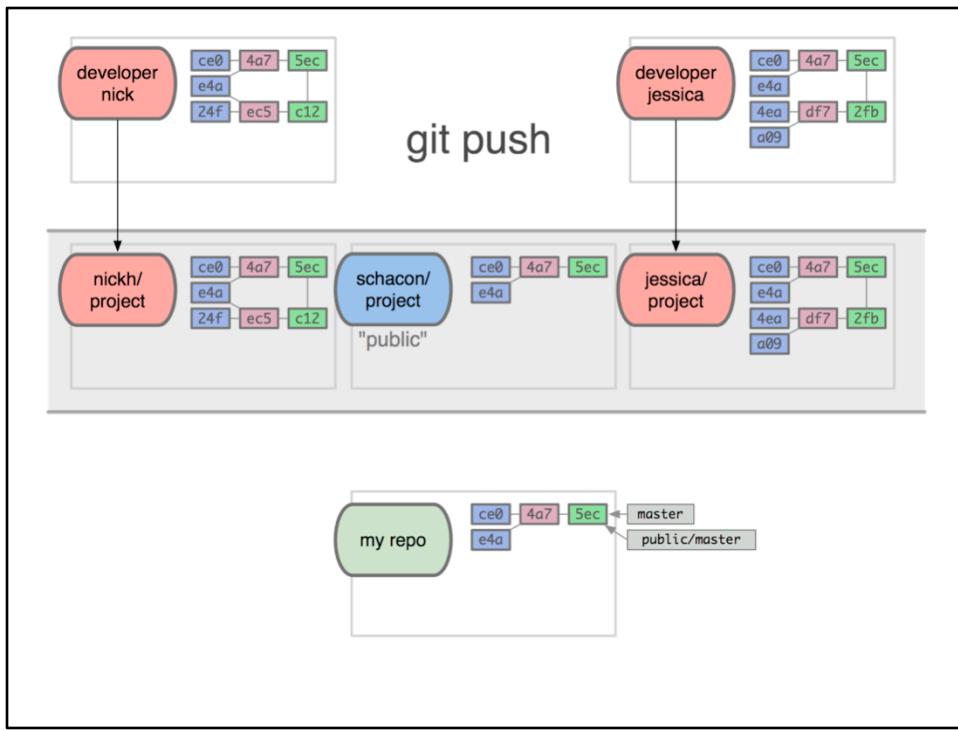
and does some work



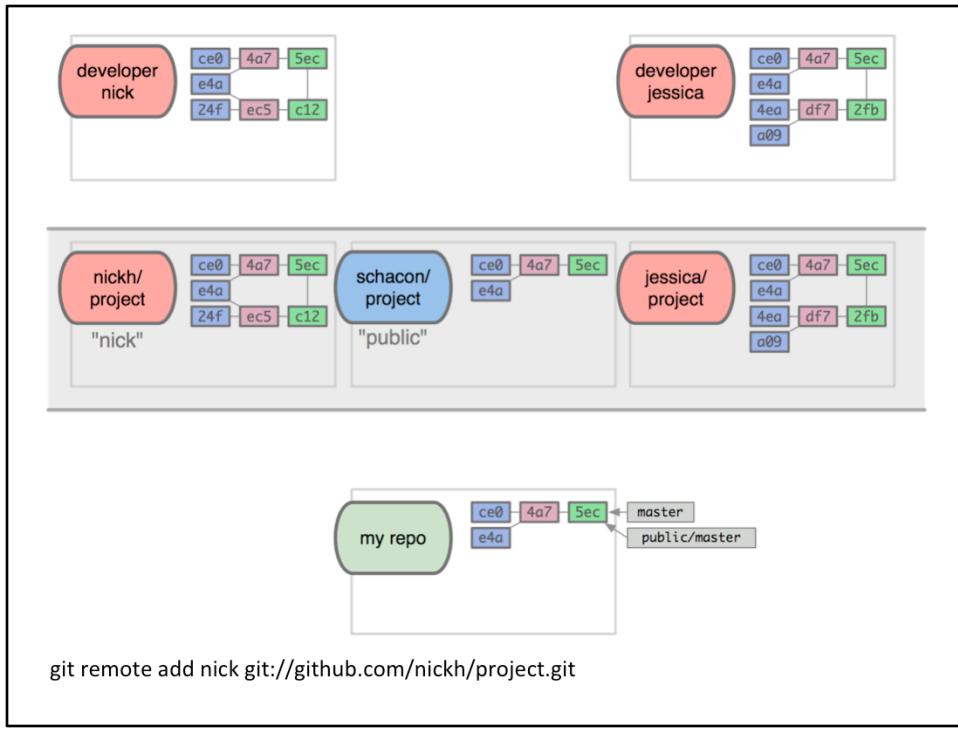
jessica clones



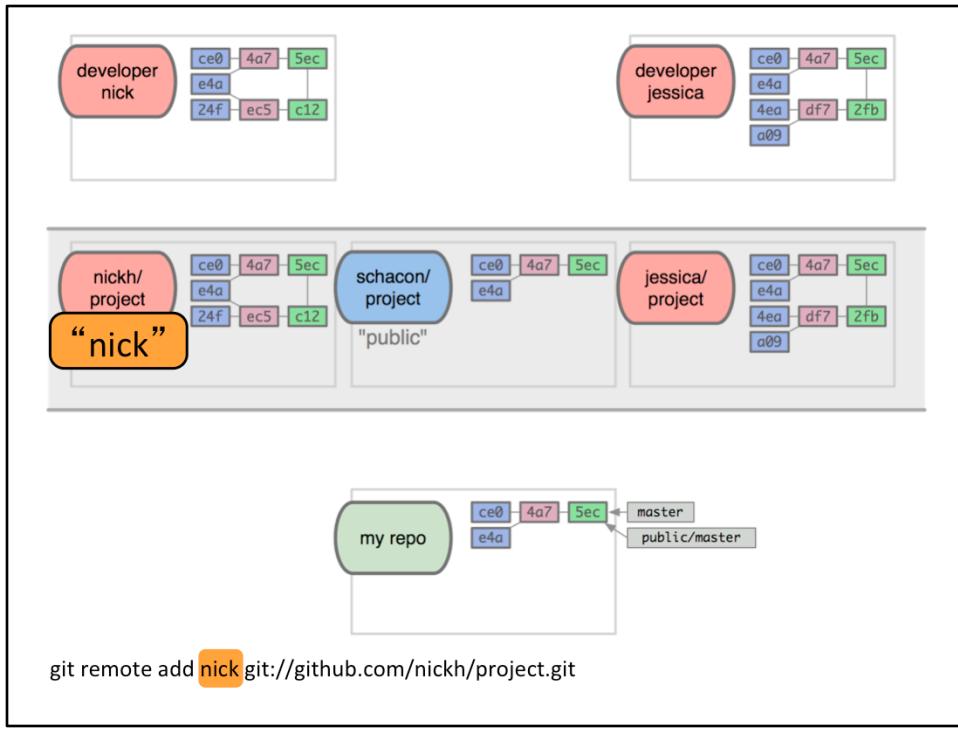
and does some unrelated work



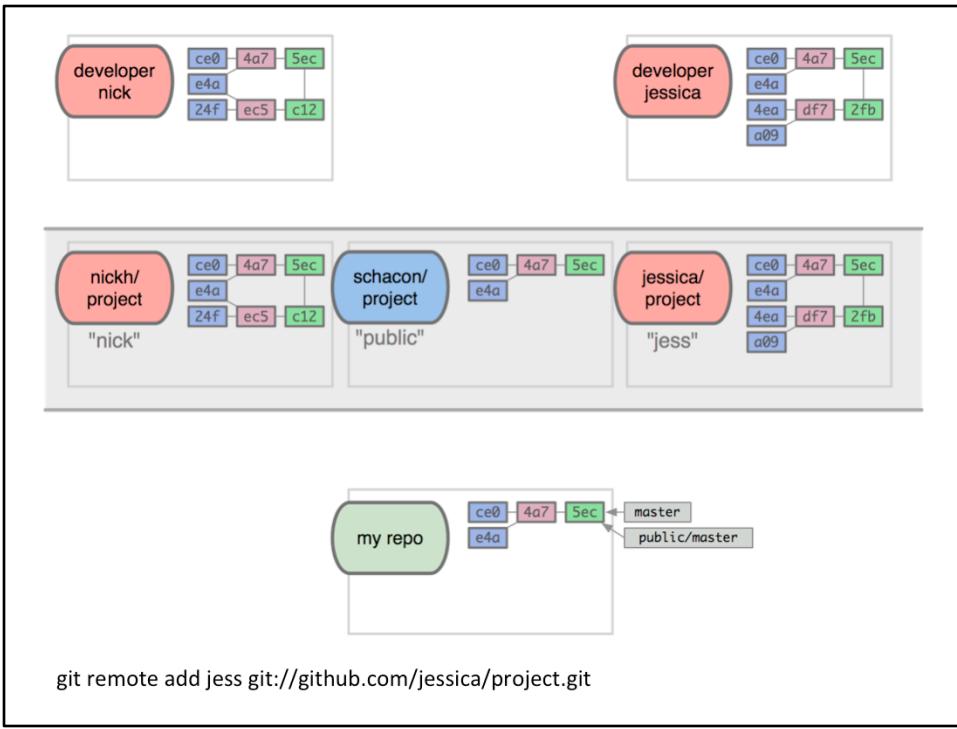
they both push to their own public repositories



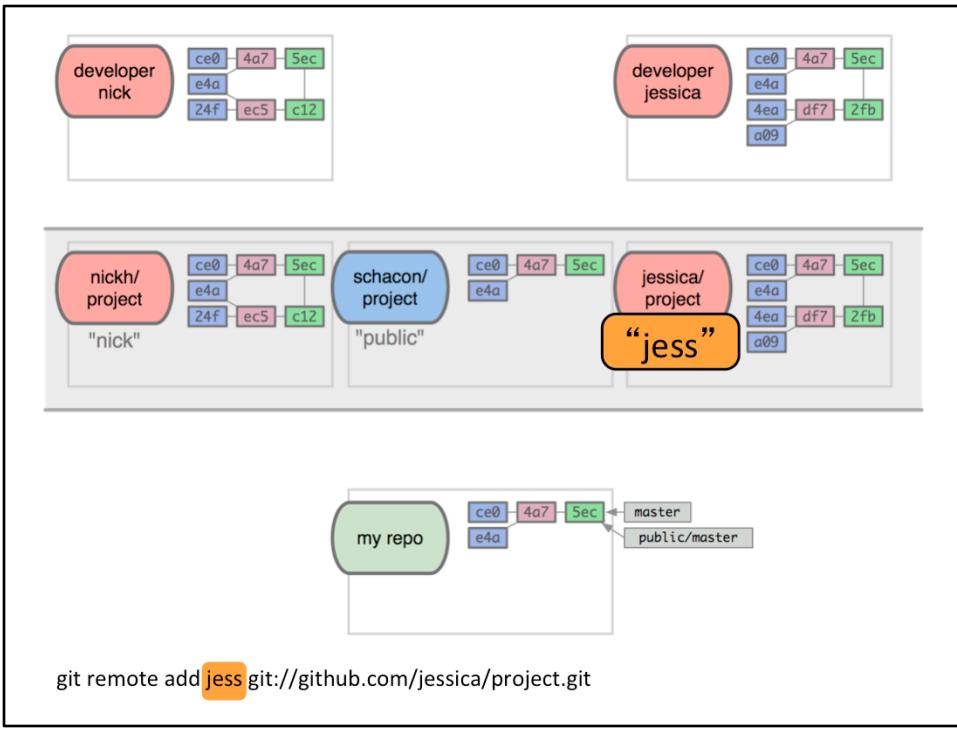
now, I'm interested in their work because it's based on mine and perhaps they *asked* me to merge it into the main project...  
 so I need to be able to access it in my local repo



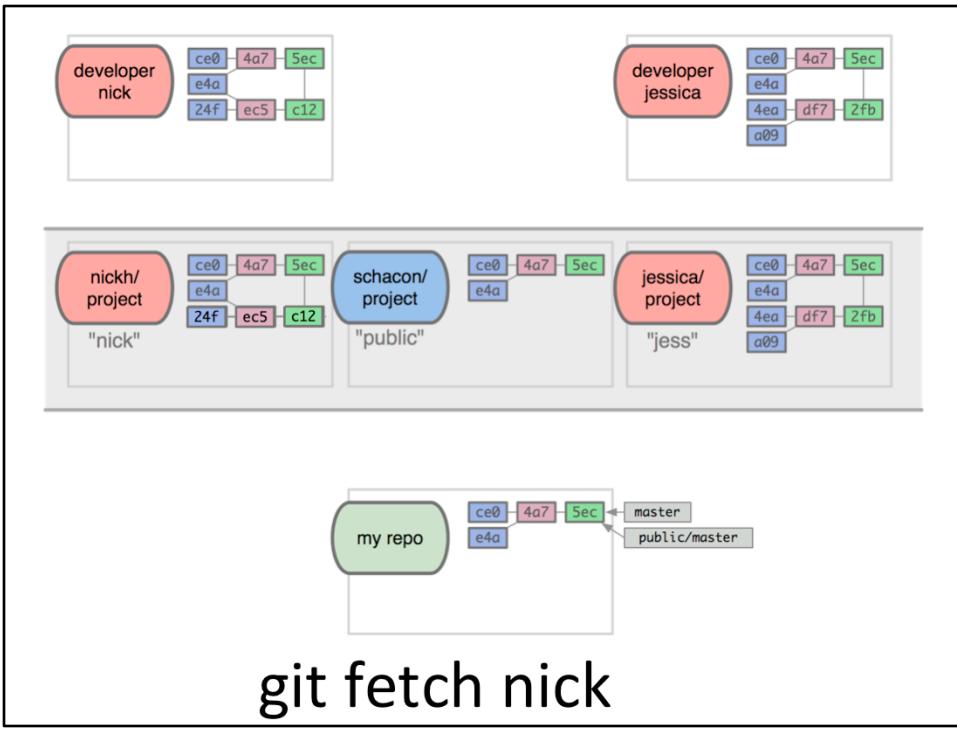
so I nickname nick's public repo url "nick"



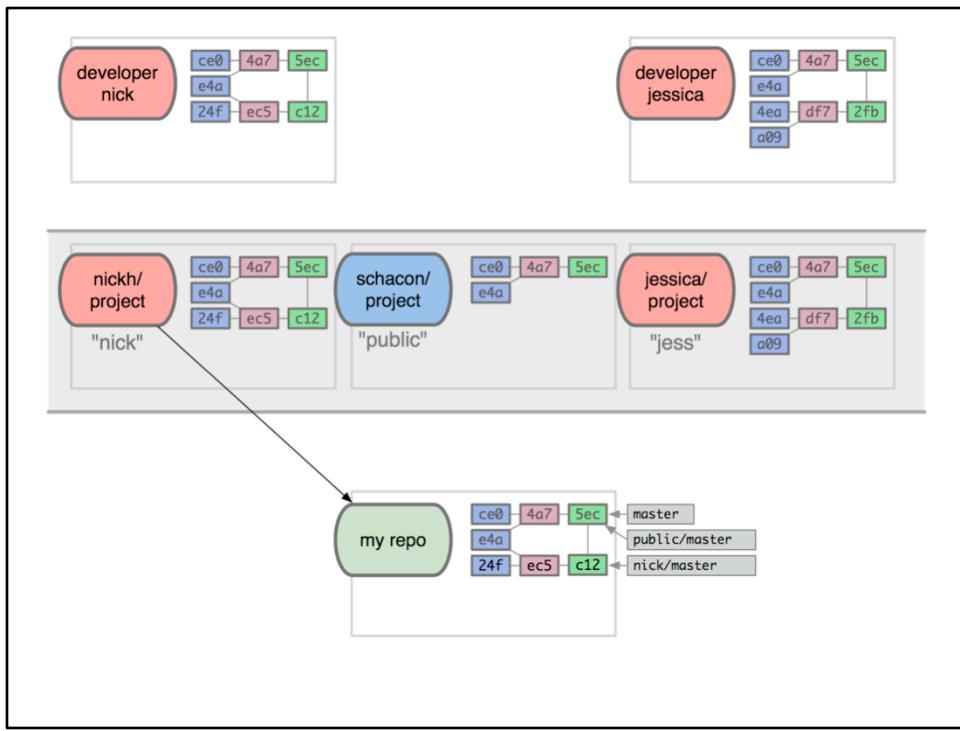
and jessica's

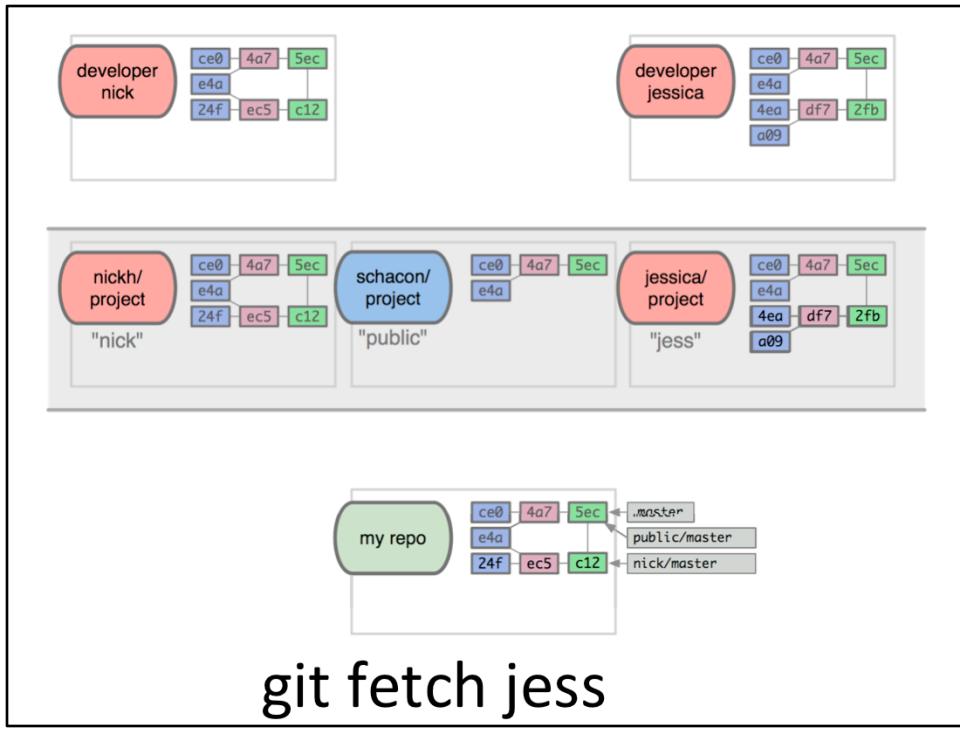


"jess"

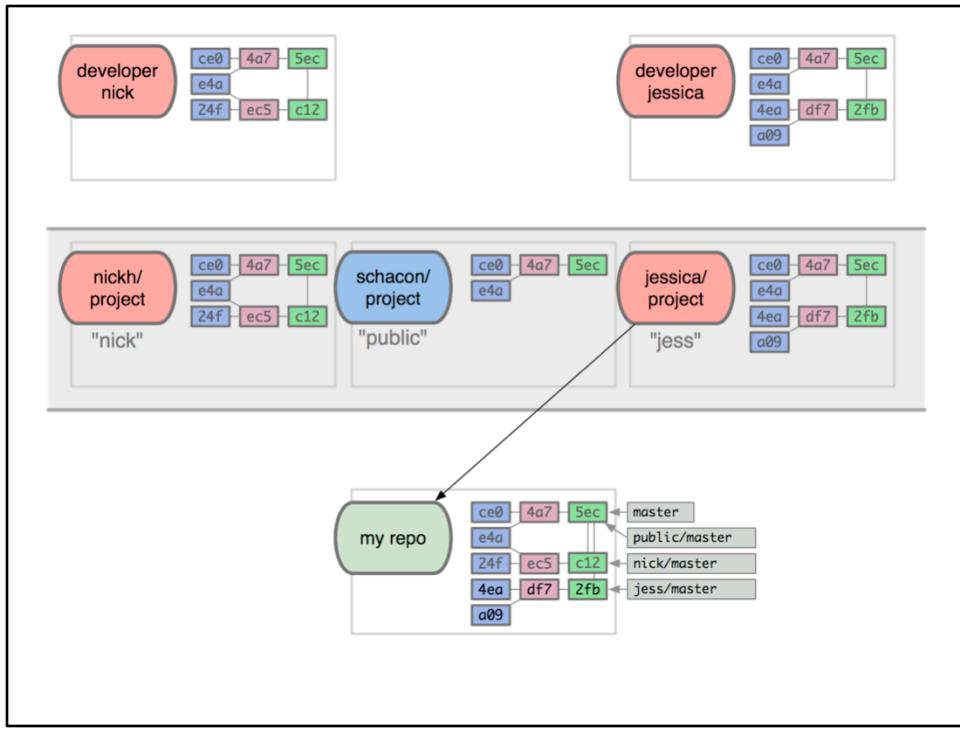


and now I can do cool things like git fetch nick

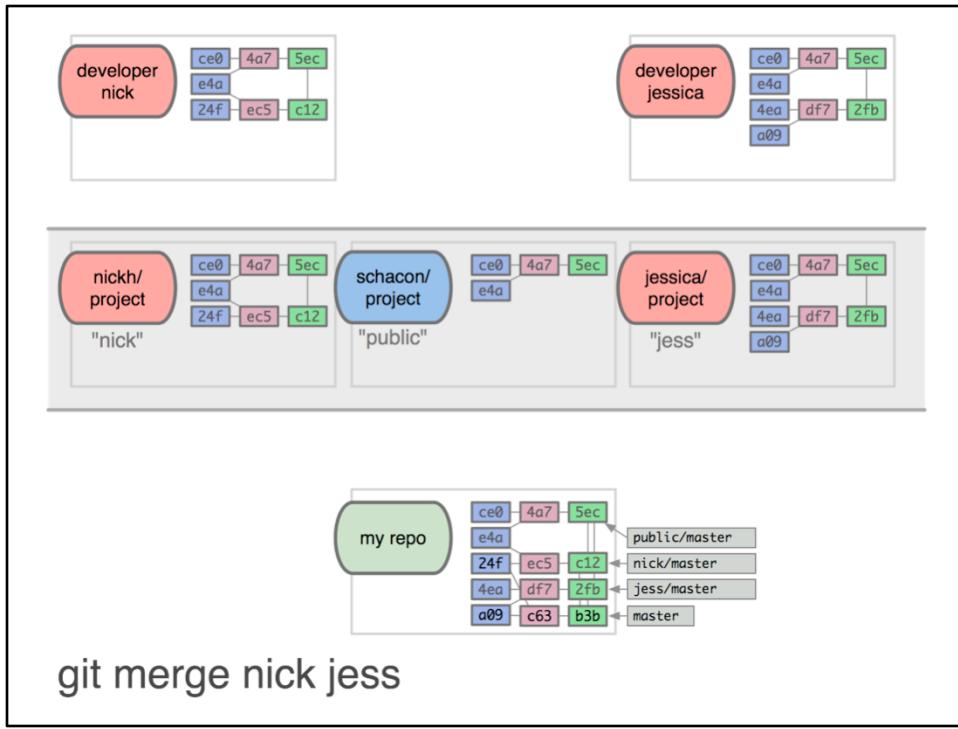




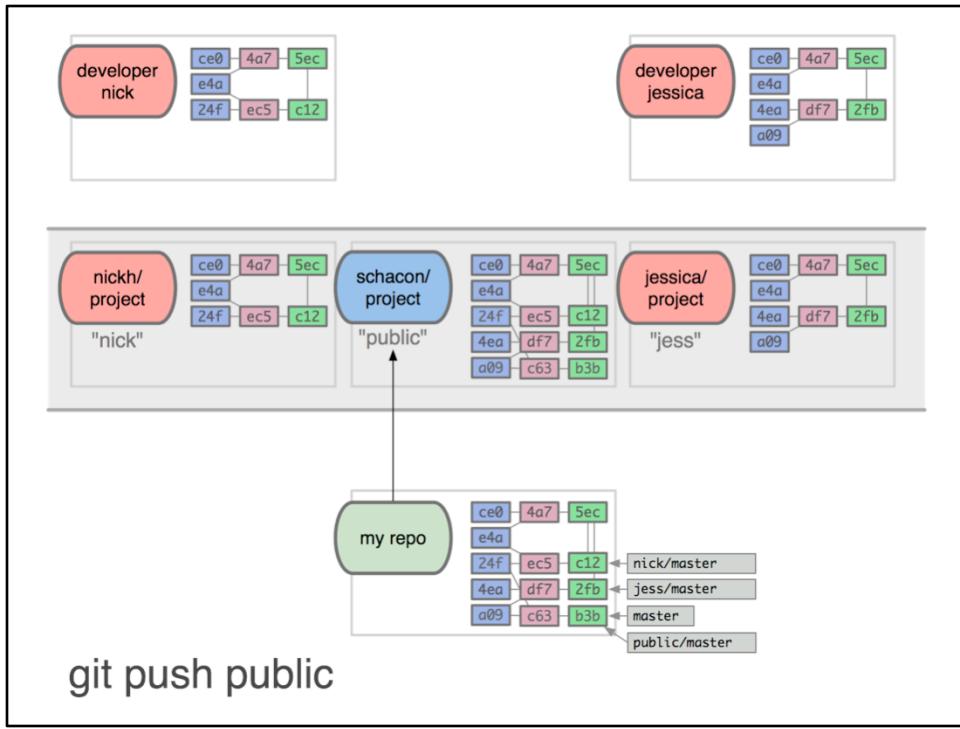
and git fetch jess



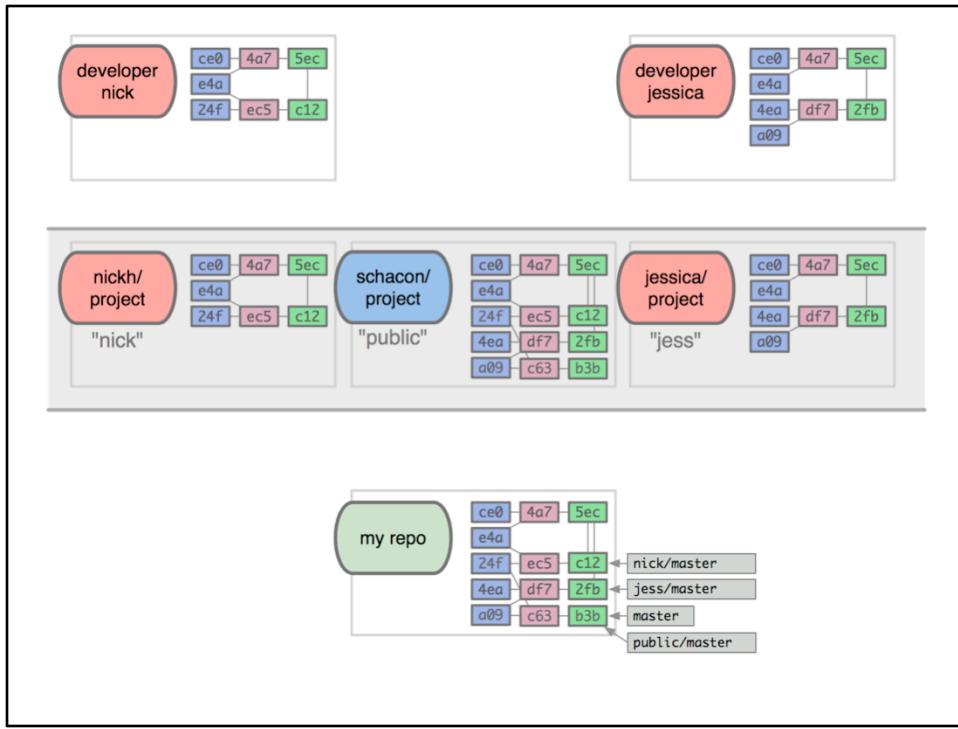
to get their work for inspection



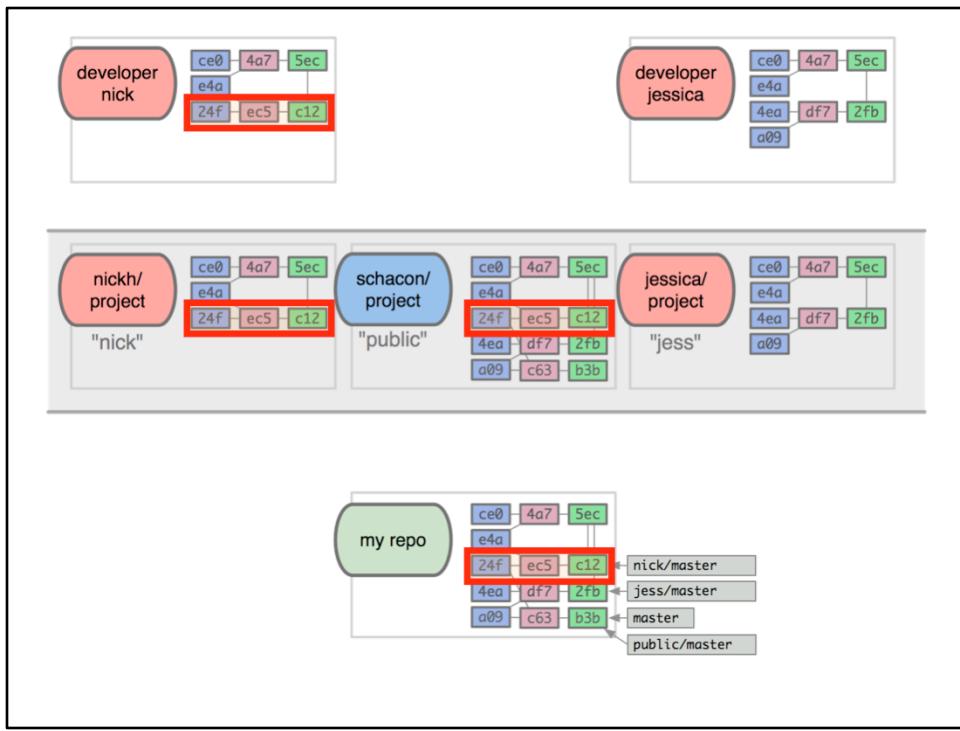
I like it all, so I do a fancy three-way merge



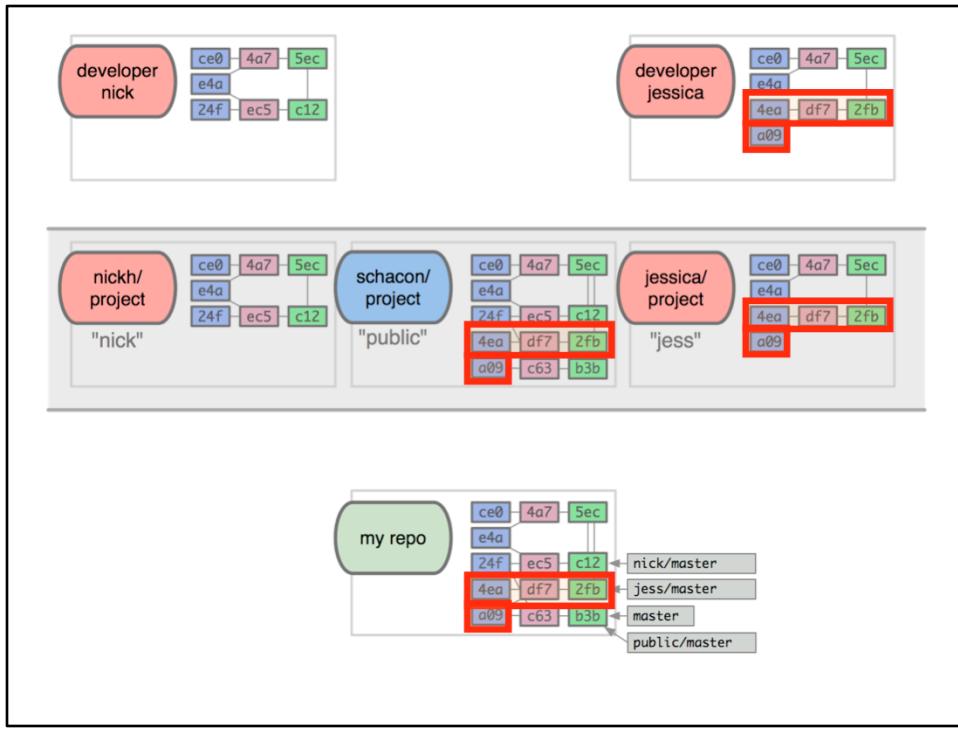
and push everything up to my public repository for the whole world to use



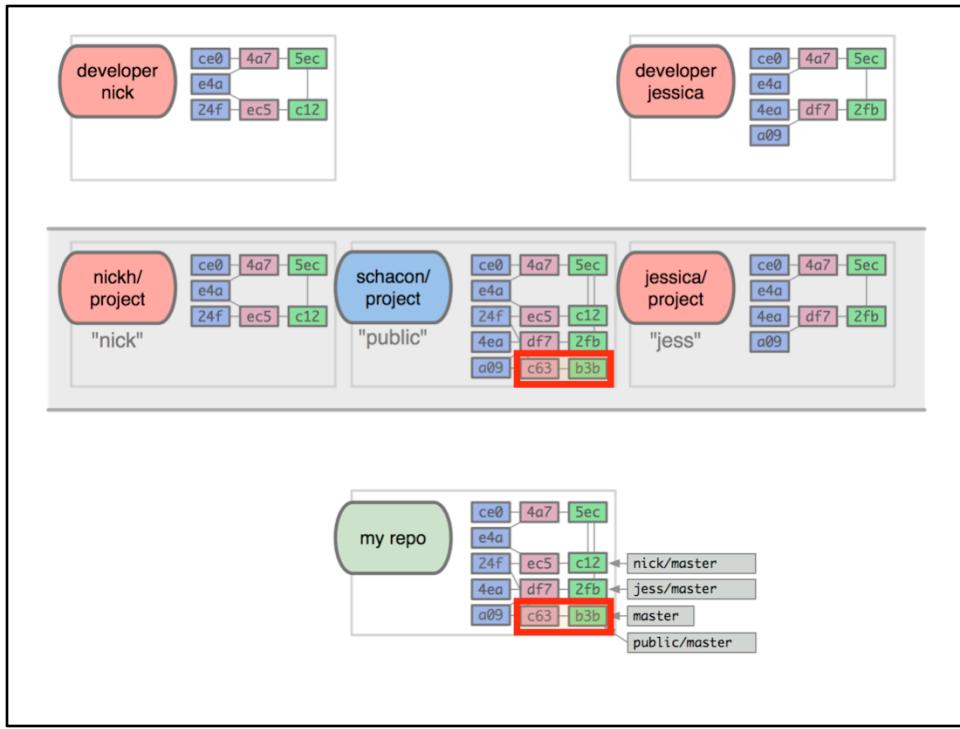
that had to be pretty fast, but notice the propagation of objects



nick's



jessica's



and my merge objects

## Understanding Git

- ~~The Basics~~
- ~~The Git Object Database~~
- ~~History Inspection~~
- ~~Branching Workflows~~
- ~~Collaboration~~
- Advanced Stuff
- Troubleshooting

that wraps collaboration

## Advanced Stuff

- the index
- hook scripts
- grab-bag of useful things
  - git add --interactive and patch adding
  - git stash [--keep-index] to tuck things away for a second
  - git cherry-pick to pluck just a few cool things someone else did out of one of their commits
  - git fsck [--unreachable] as the final defense against lost stuff
  - git reflog to see the true history
  - git bisect to find out when an error was introduced

on to advanced stuff, starting with...

## The Index



the index  
until now, I have failed to discuss one  
of the most interesting and  
misunderstood features of git

# The Index

```
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
│   ├── ...
├── index
├── info
│   └── exclude
├── logs
│   ├── ...
├── objects
│   ├── ...
└── refs
    ├── heads
    │   └── master
    └── tags
        └── taggy
```

15 directories, 22 files

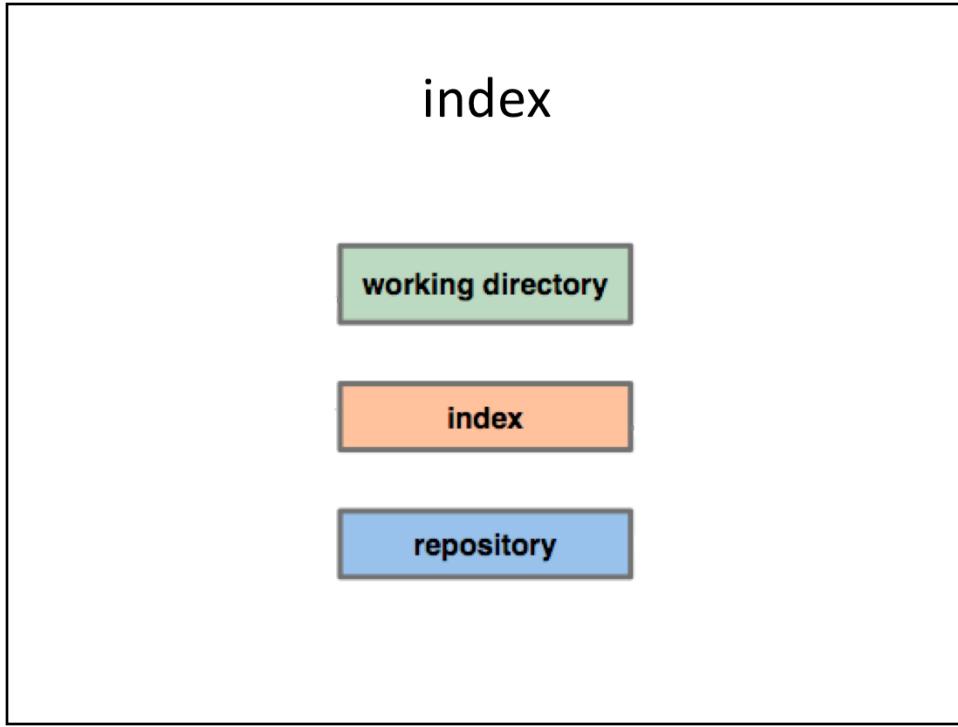
you remember the .git directory

# The Index

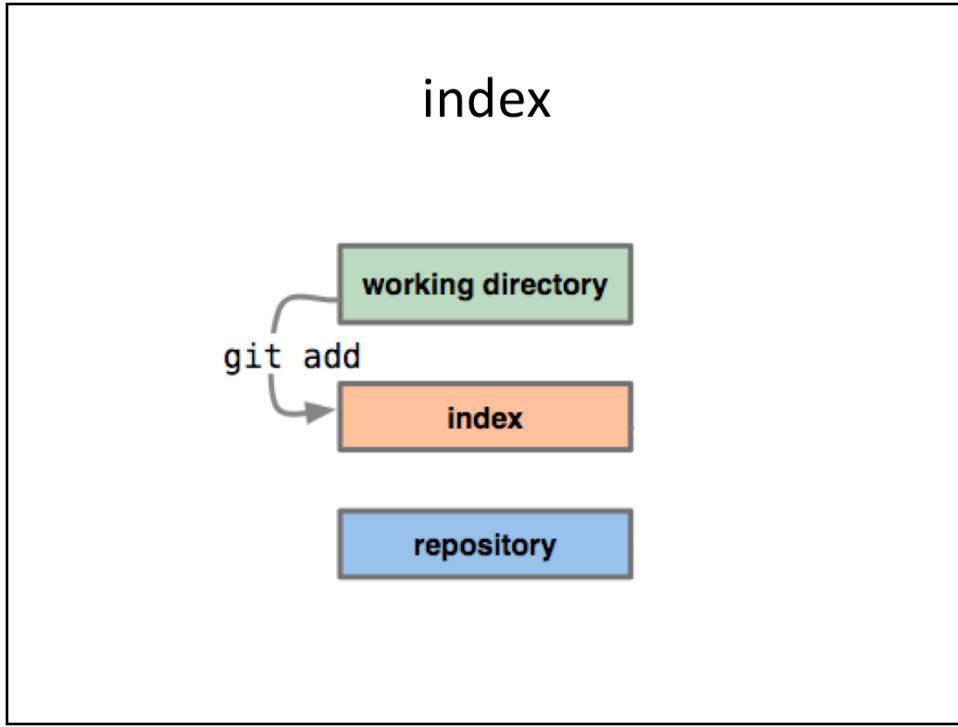
```
.git
├── COMMIT_EDITMSG
├── HEAD
├── config
├── description
├── hooks
│   └── ...
├── index ← there it is
├── info
│   └── exclude
├── logs
│   └── ...
├── objects
│   └── ...
└── refs
    ├── heads
    │   └── master
    └── tags
        └── taggy
```

15 directories, 22 files

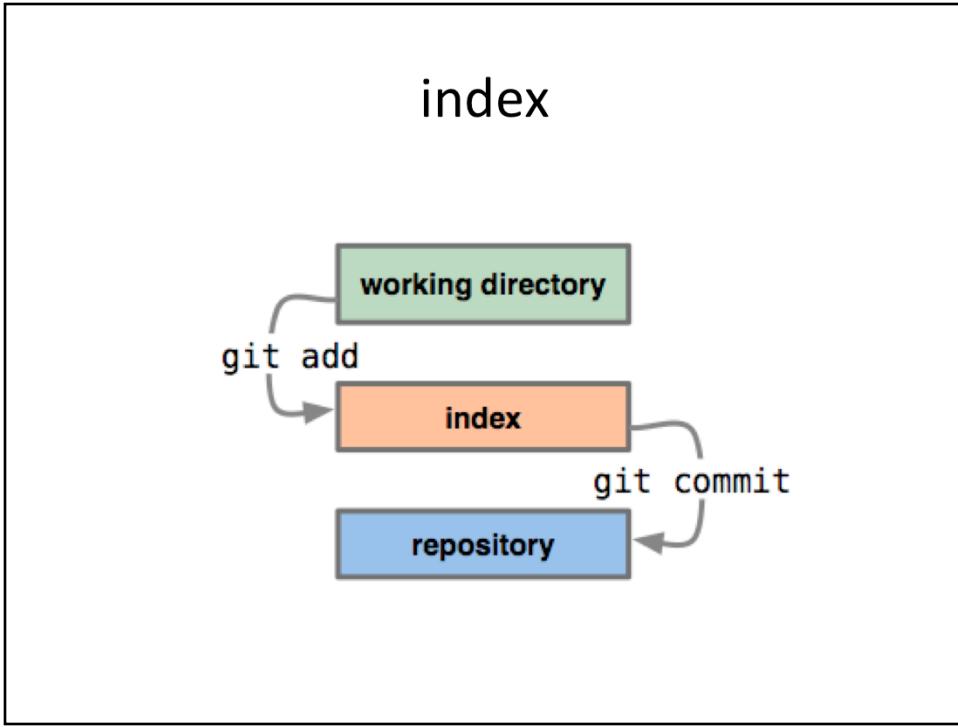
well, there's the index



the index is a cache that sits in-between your ever-changing working directory, and your permanent repository



when you run ‘git add’ on files, it caches the snapshot of *those* versions of the files in the index



then when you run ‘git commit’ , it actually writes the commit object pointing *not* to the versions of the files in your working directory, but what they looked like when you ‘git added’ them

what?



imagine we have some repository, and its state is laid out here



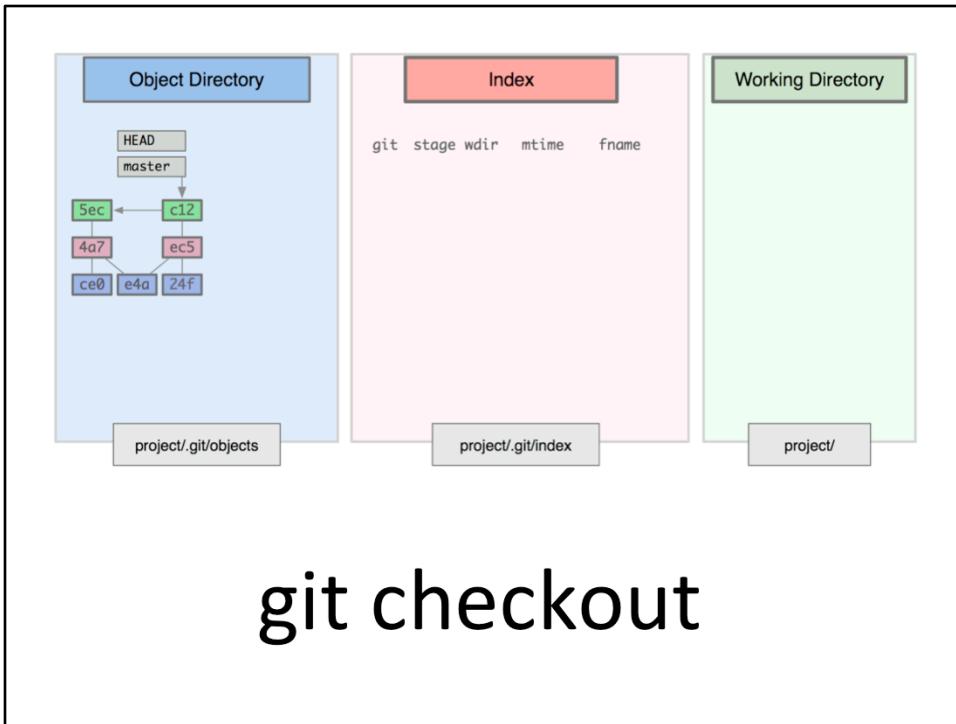
on the far left we have the git object database, which currently contains two commits, one tree and two blobs



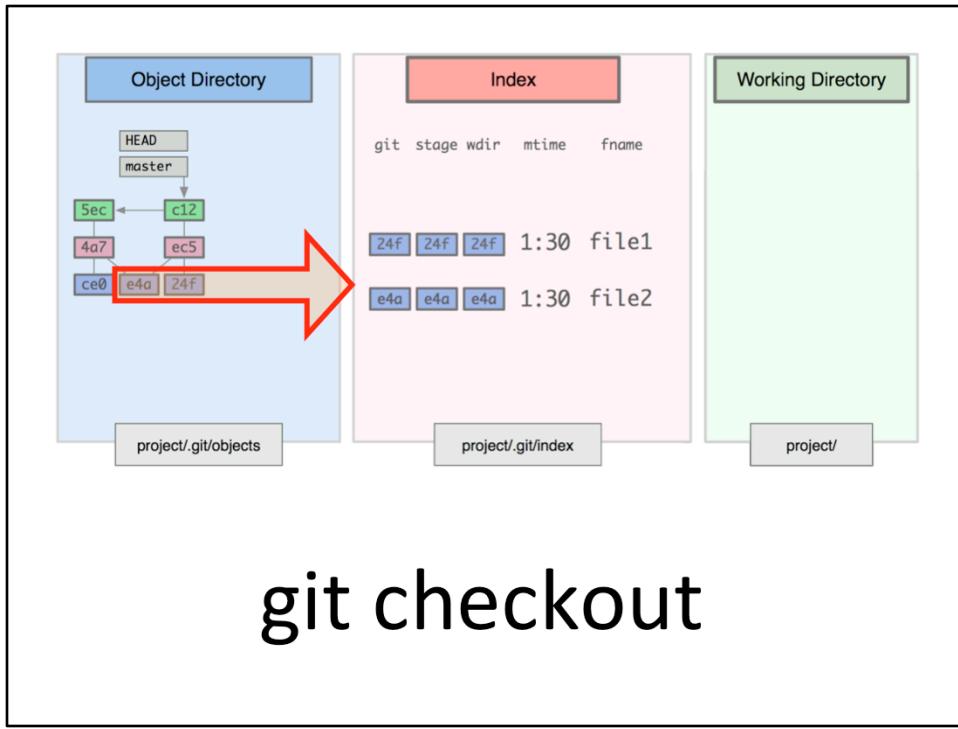
on the far right our working directory,



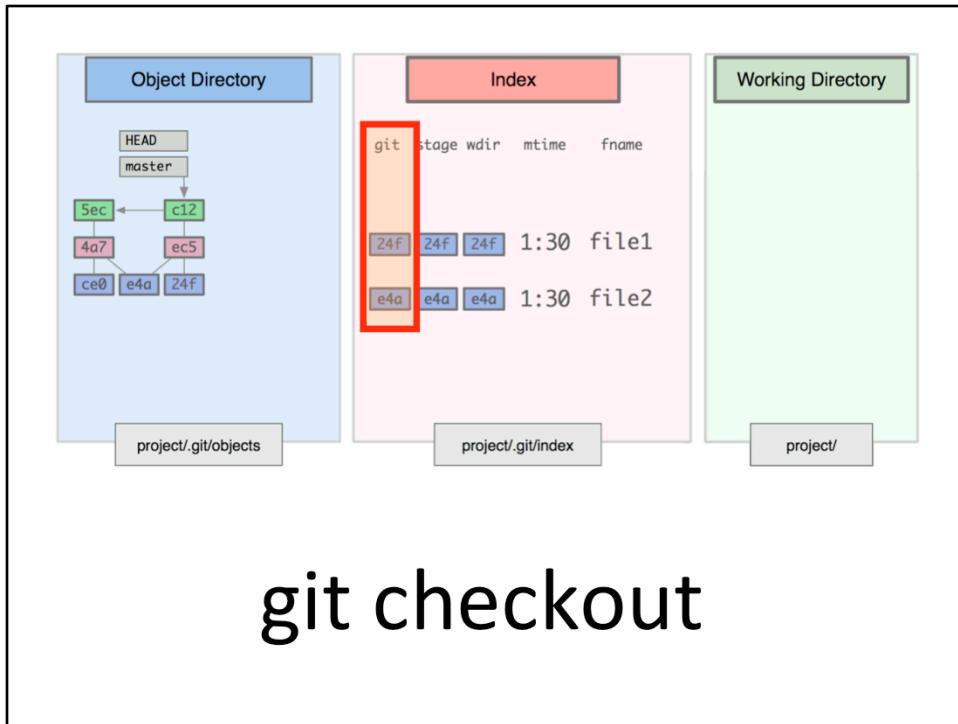
and in the middle the index.



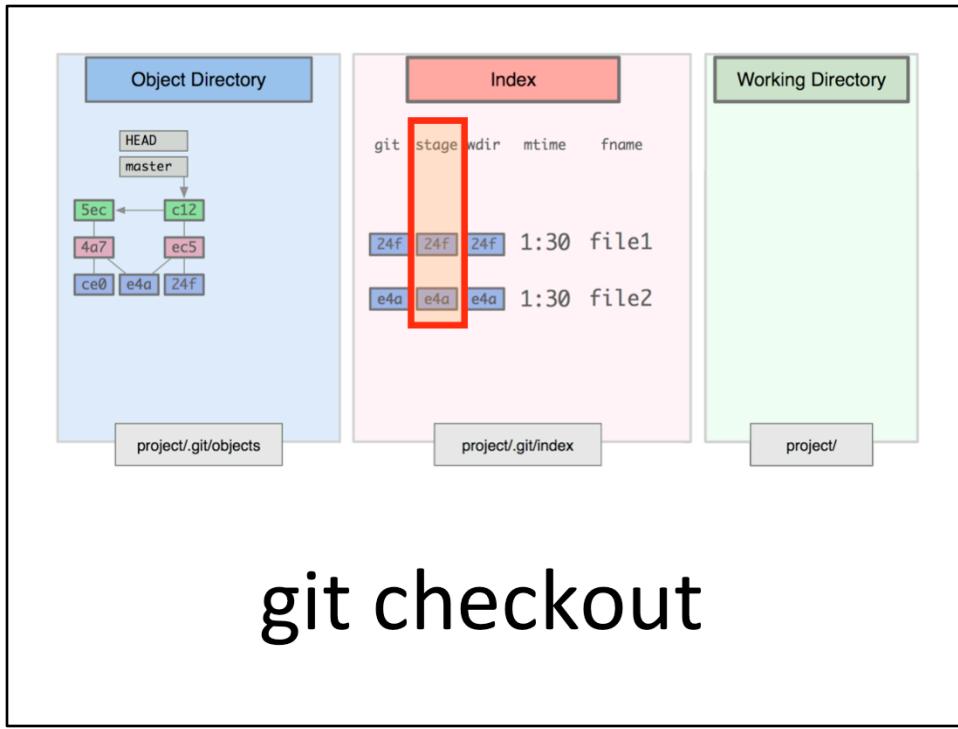
we then checkout master,



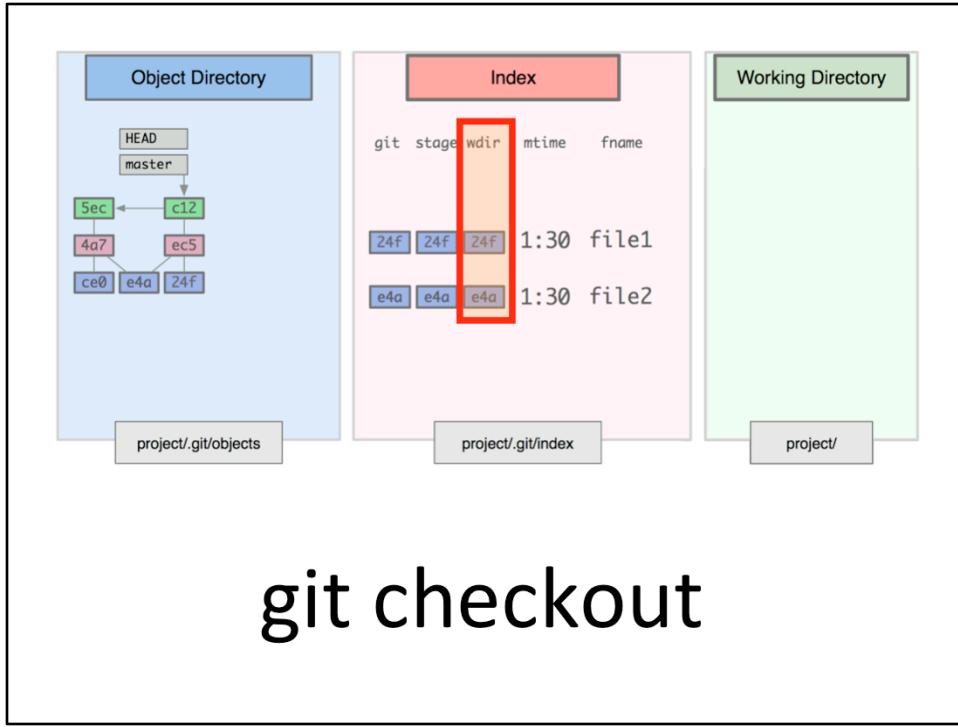
which populates the index with



the SHAs of the files as they appear in the object database,

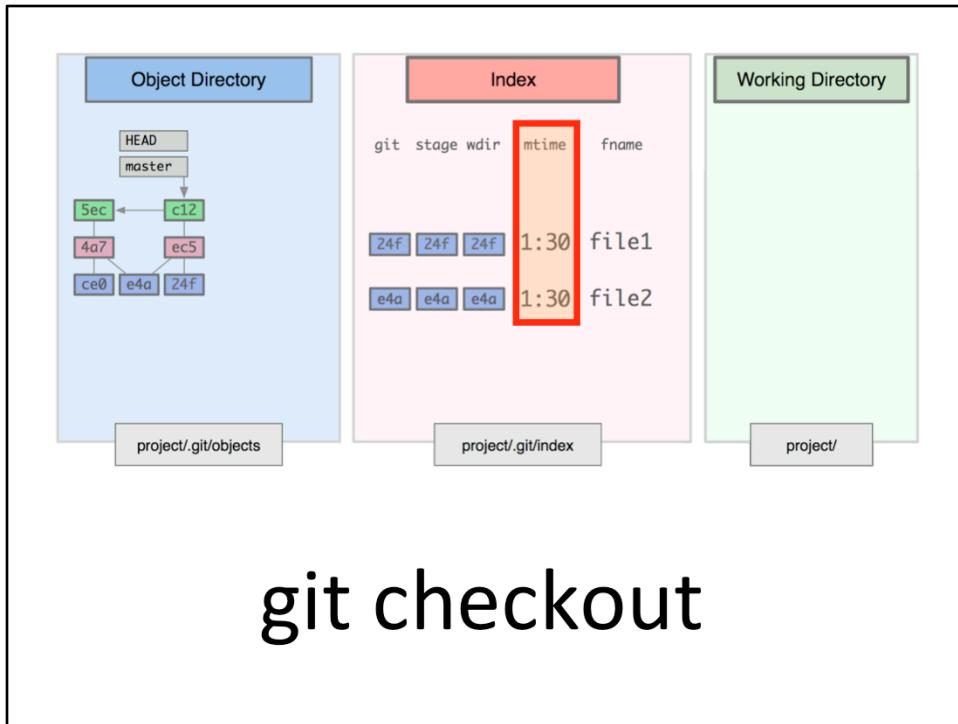


the SHAs of the same files as they are in the “staging area”,

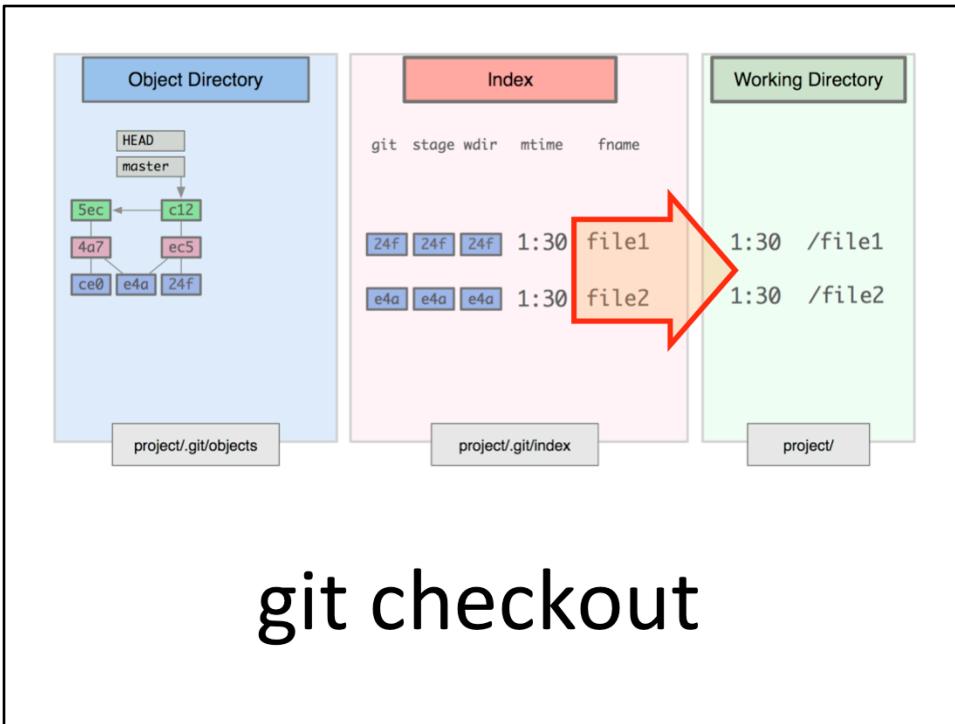


and the SHAs as they are in the working directory.

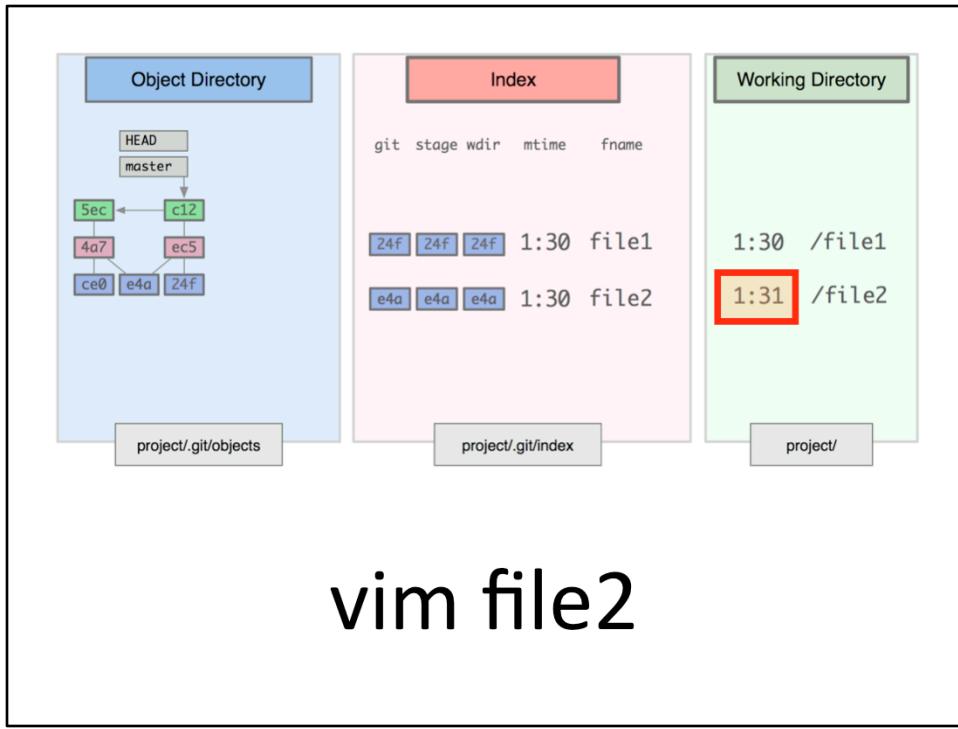
(and by the way, the SHAs are always 40 characters, but I use just three here to save space)



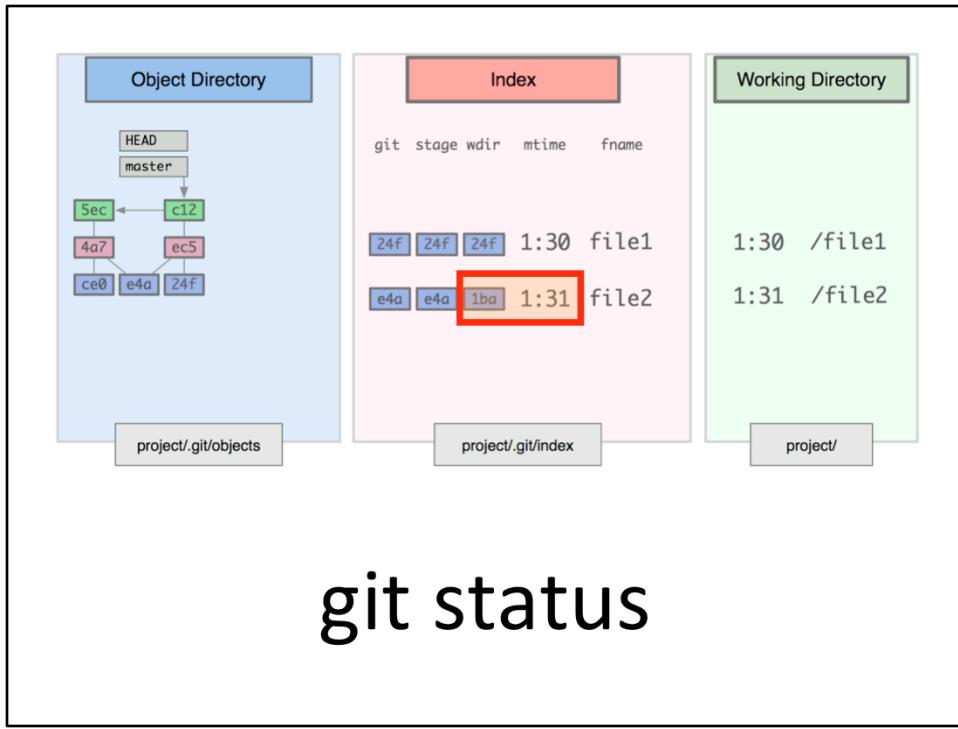
it also keeps track of the modified time



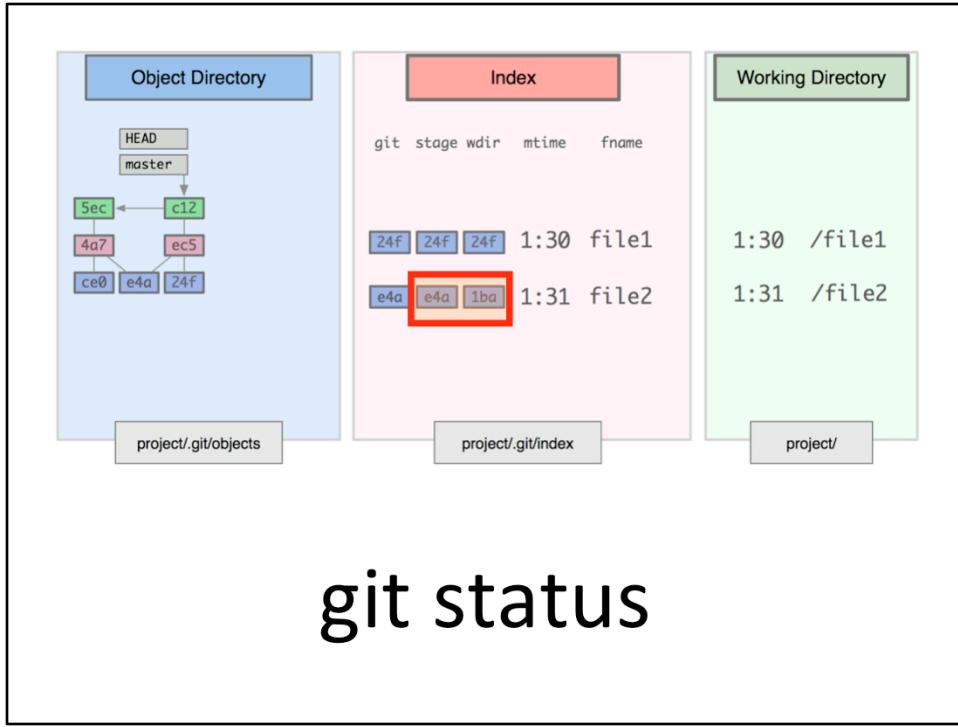
and the filenames, which both match those in the working directory.



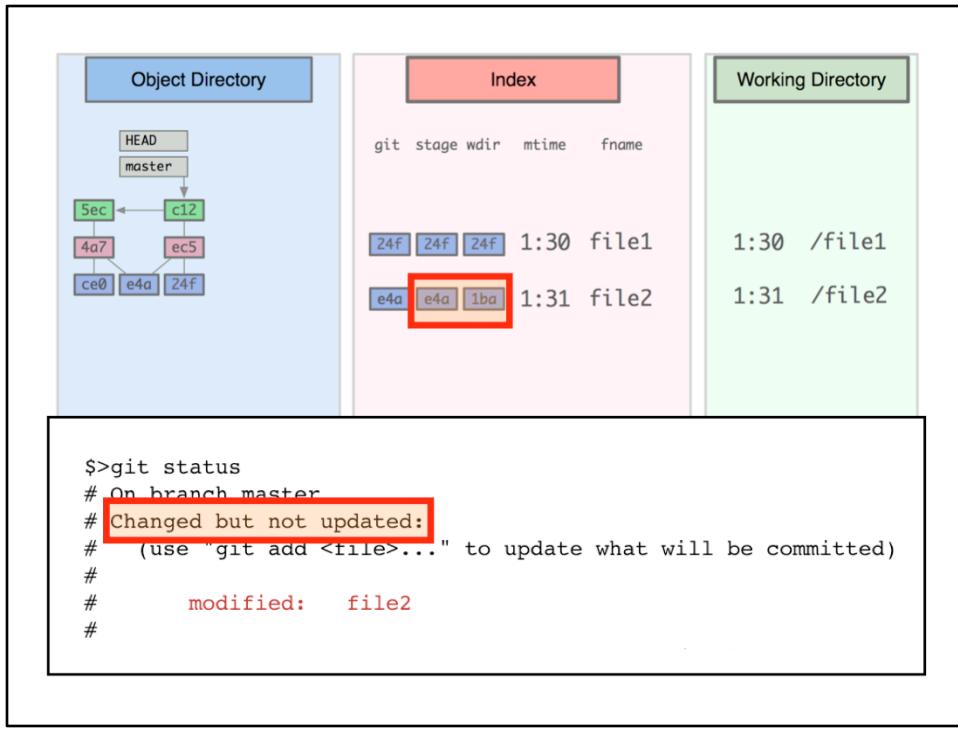
now say we edit file2, which changes the mtime



when `git status` is run, the index is updated with `file2`'s new SHA and mtime under the working dir column

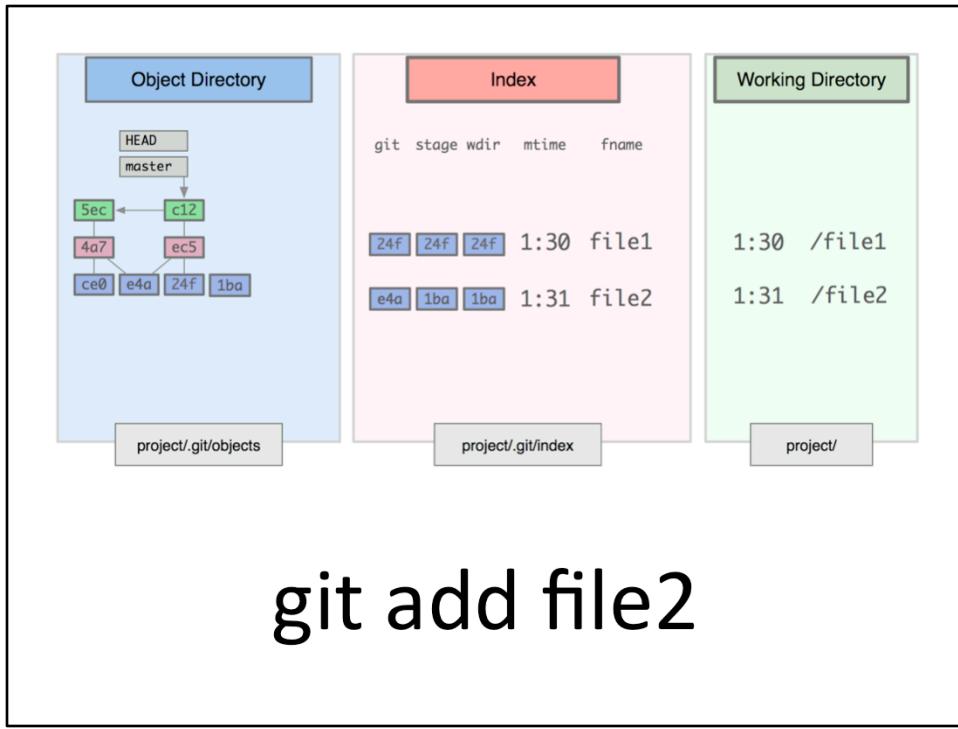


because the “staged” SHA for this file differs from the working dir SHA,

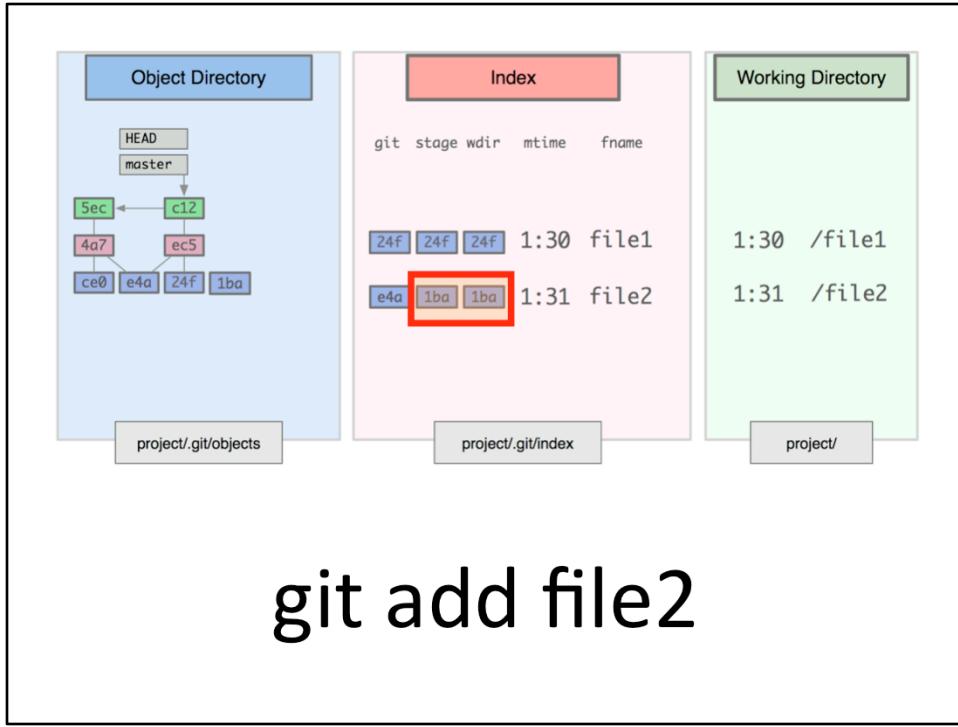


git knows to tell you that you have a file that has been “changed but not updated”

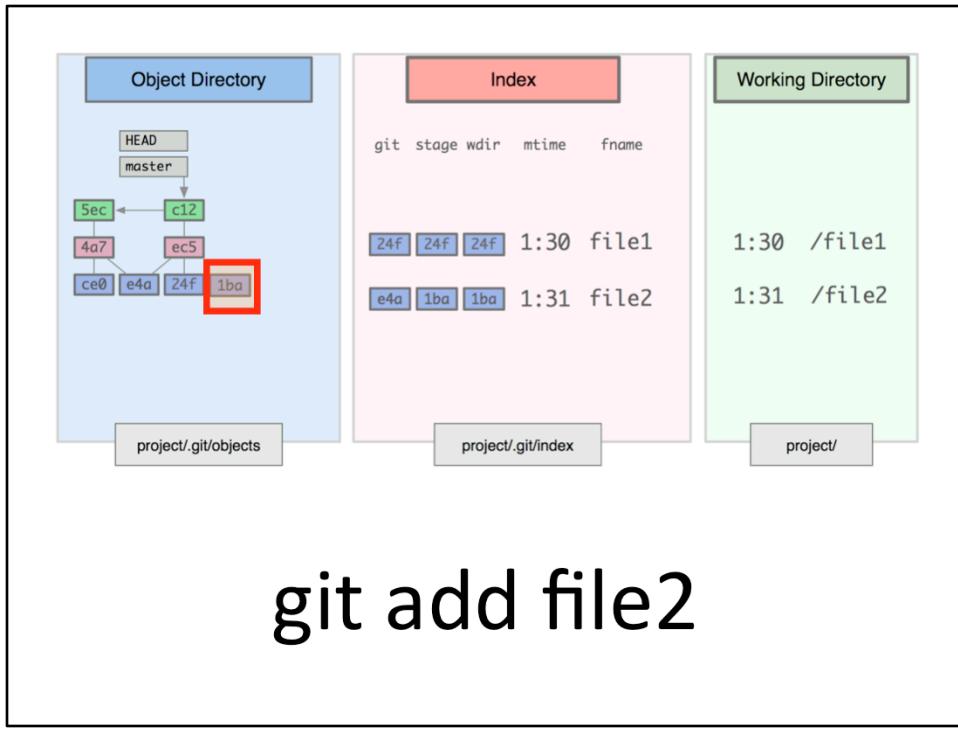




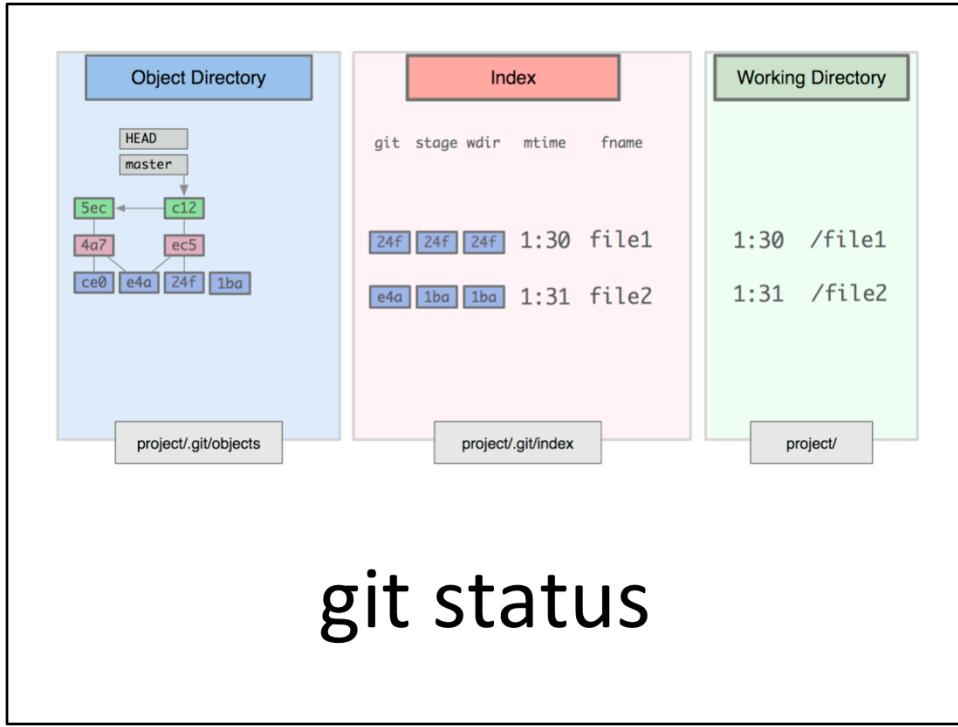
now let's "add" that changed file to the staging area  
 people usually just say that the file is "added to the index", or "staged"



notice that after an add, the staged SHA for the file matches the working dir SHA, which is still different from the object directory's SHA because the file hasn't been committed



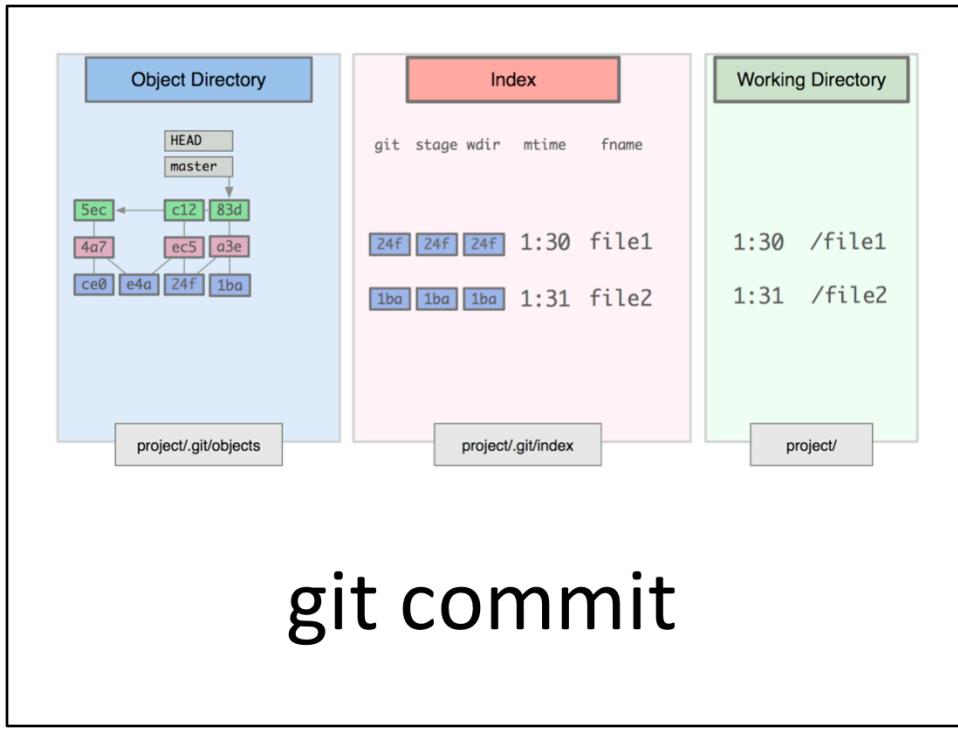
however, a dangling blob of that file's *current* state *has* been added to the object database before any commit



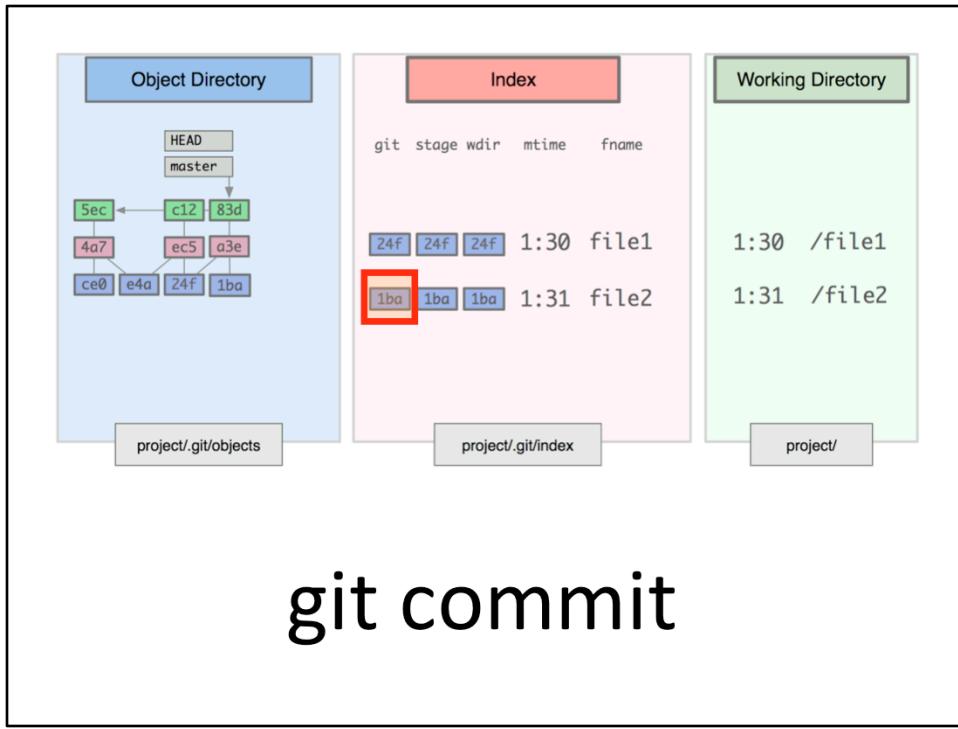
now if you run `git status`,



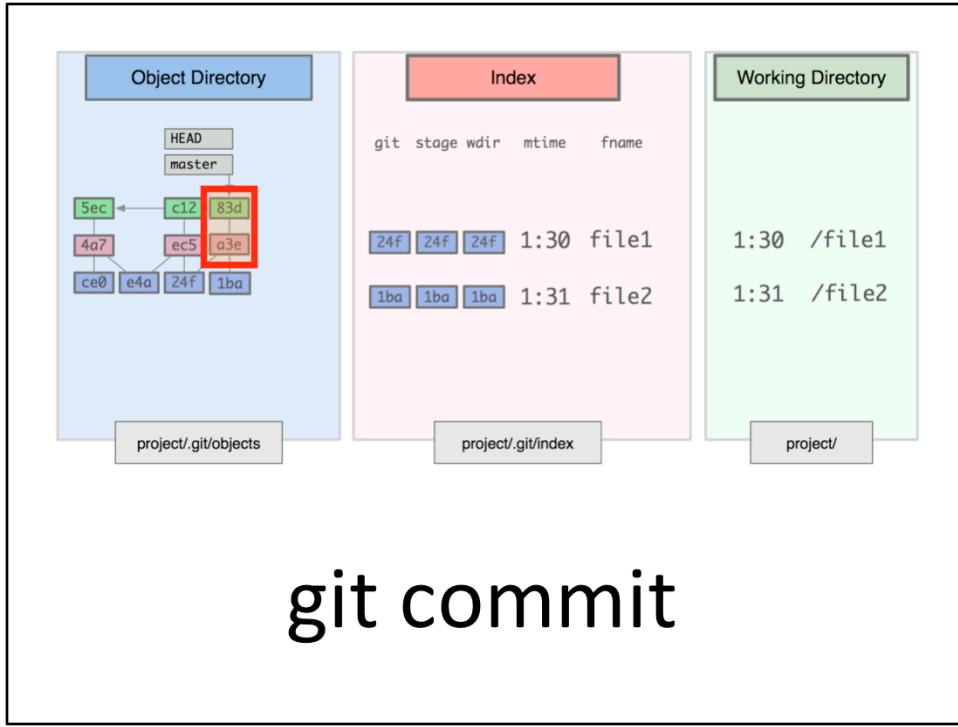
git knows that the file is staged to be committed.



so when you run `git commit`



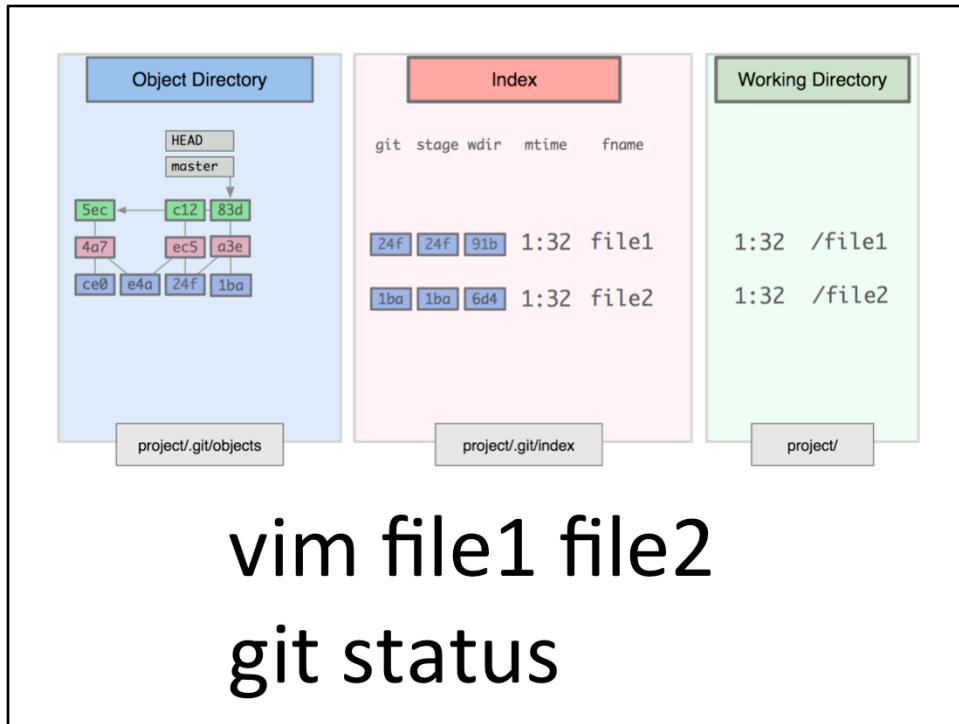
it updates the index appropriately,



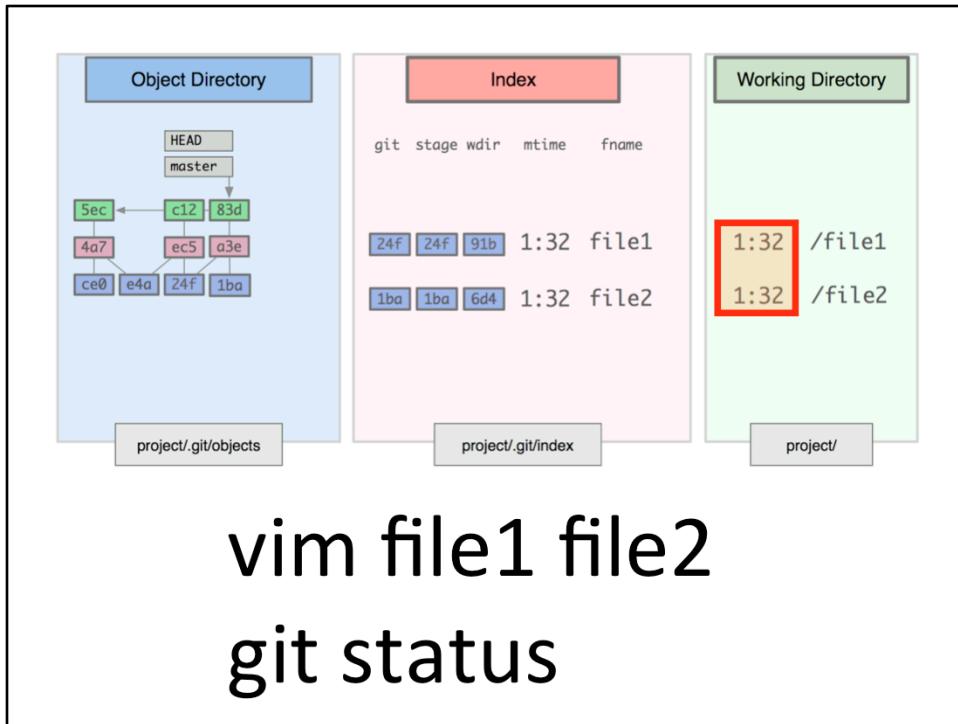
writes the new commit in the object database, the new tree containing one old file blob and the one *new* file blob that was stashed there earlier,



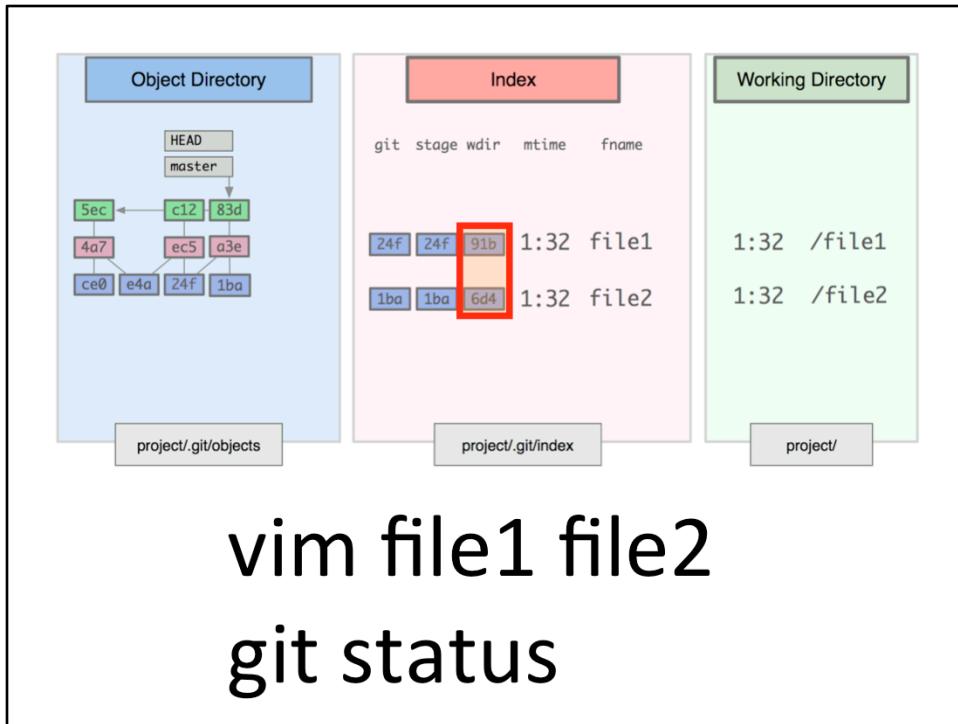
and moves the branch forward to the new commit.



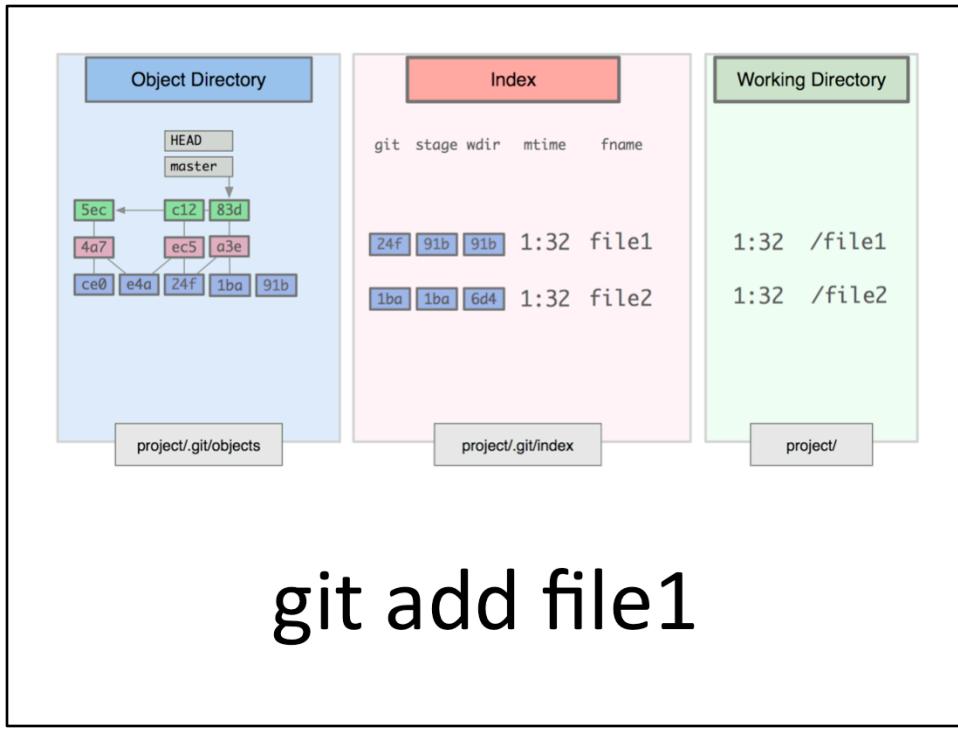
one more time.  
we edit both files again, and run git status.



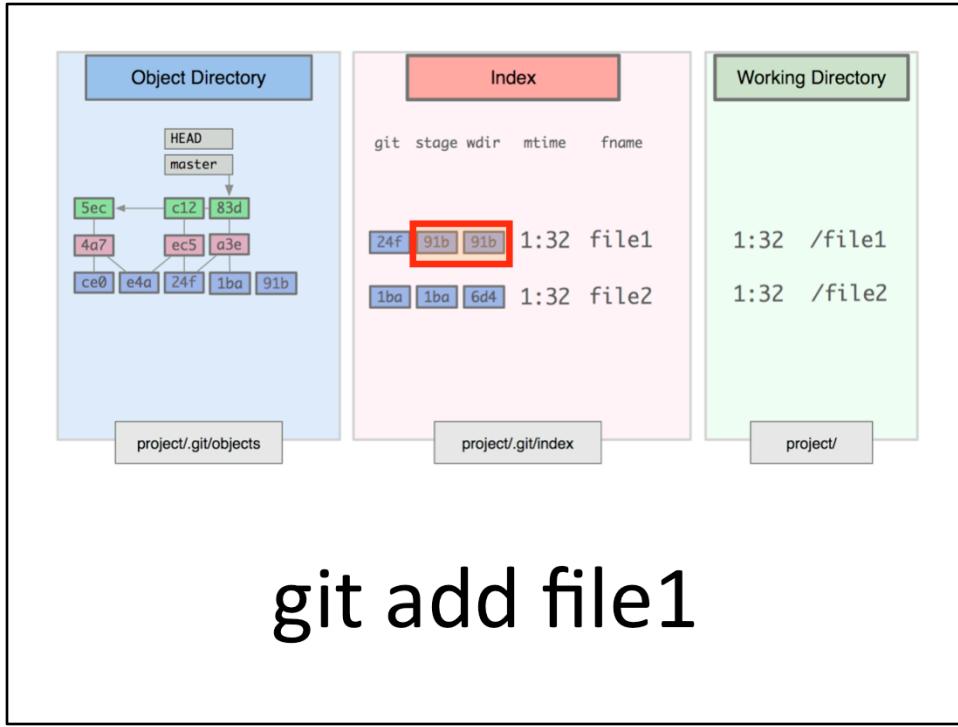
the mtimes are both different,



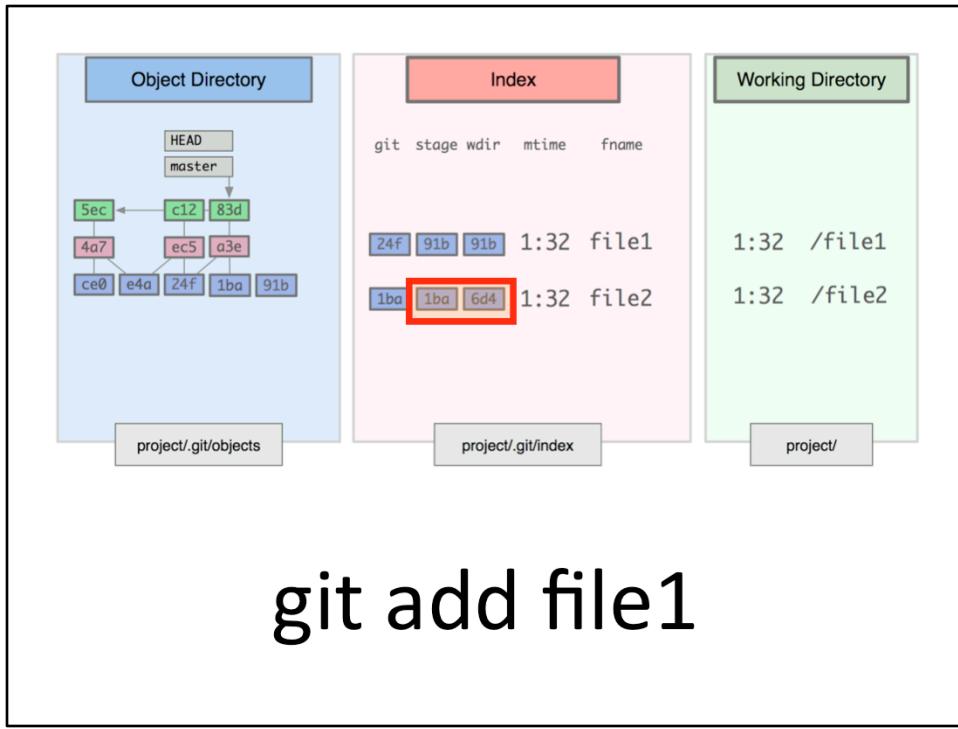
so git knows to update working dir SHAs.



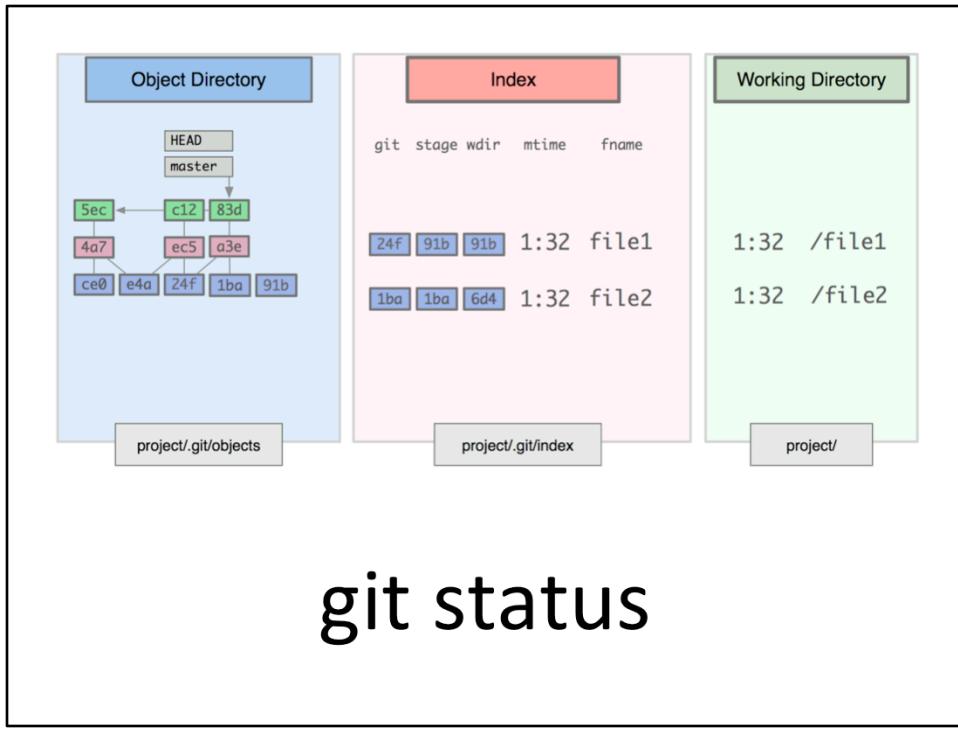
this time we'll only git-add file1,



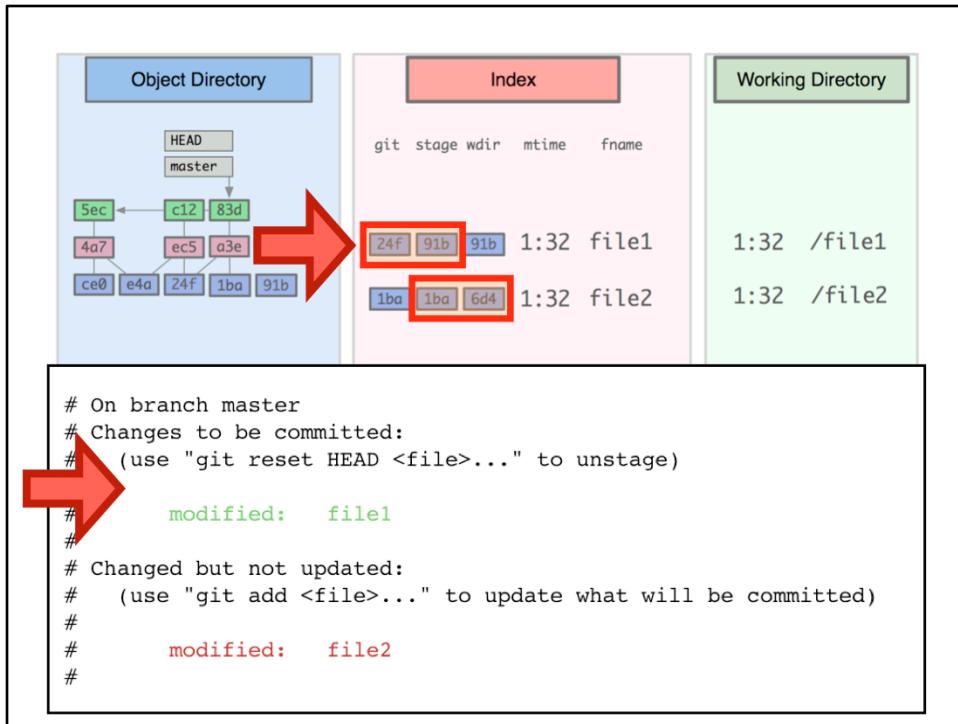
which updates only *its* staging column in the index to match the working dir,



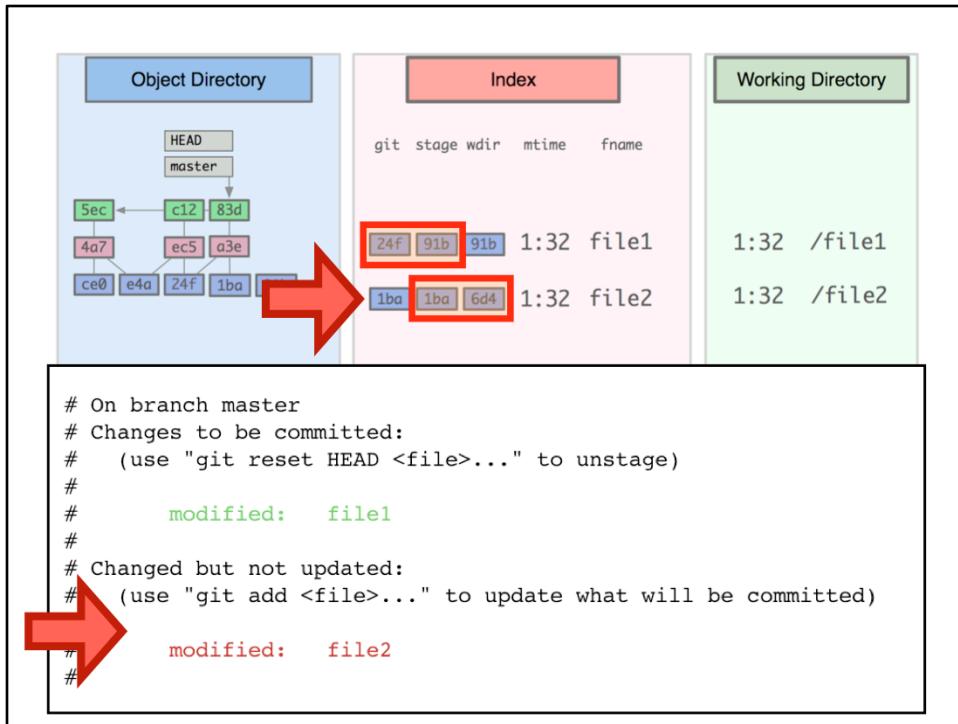
and not that of file2.



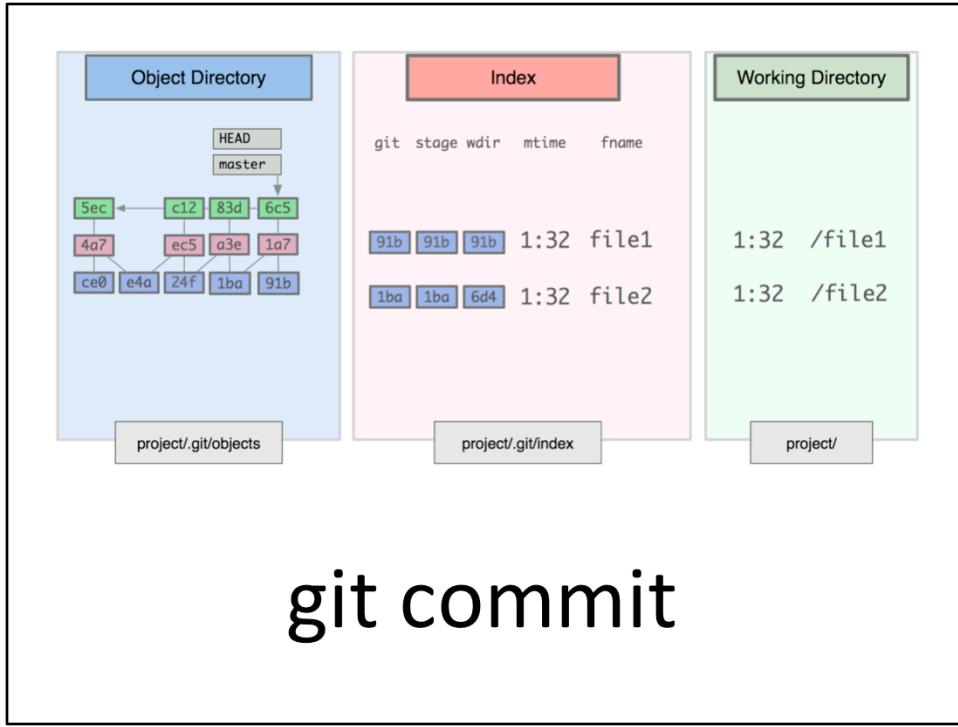
so when we run git status



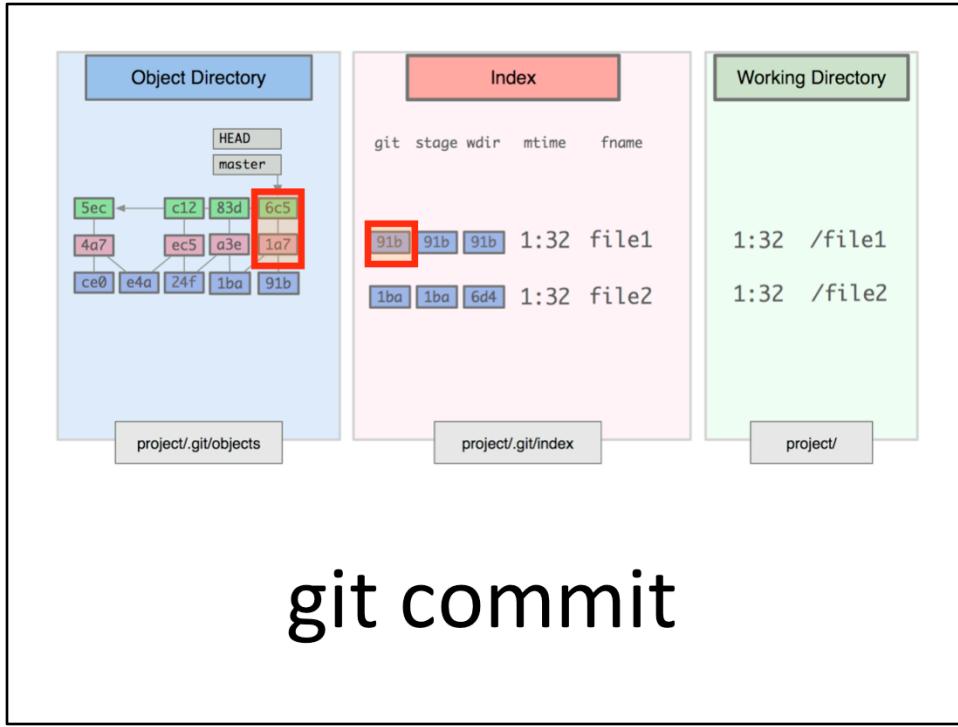
git tells us the right thing about both files, based on the differences between the relevant columns in the index.  
 file1 is a change to be committed,



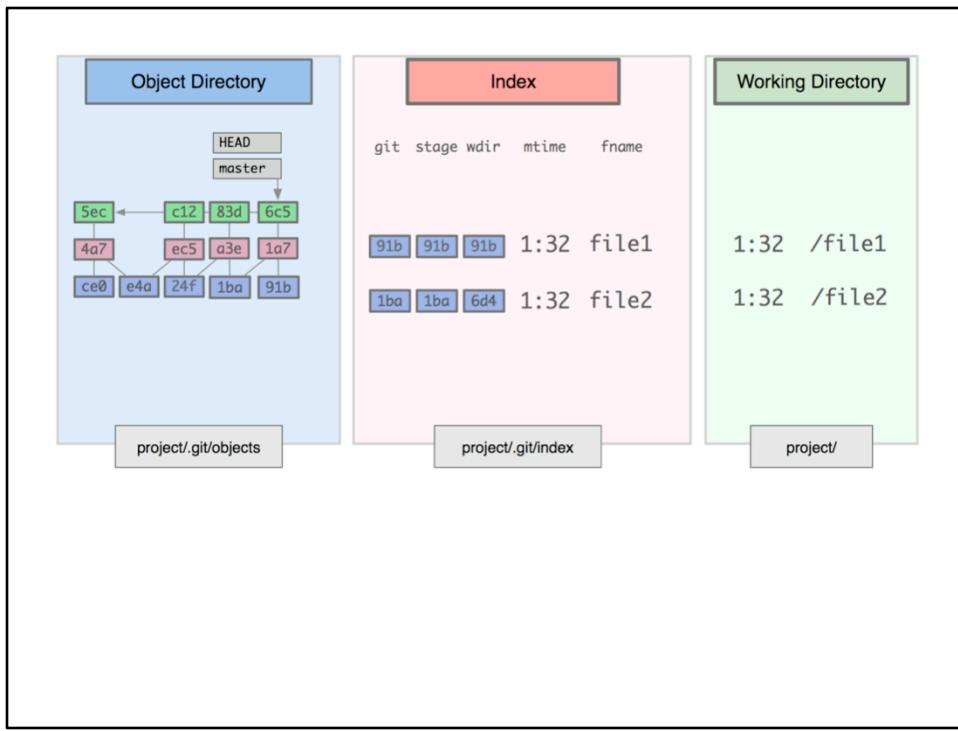
and file2 is changed but not updated.



and of course, when we then commit,



the index is updated and the object dir receives a new commit and a new tree with the new file1 blob, and the master branch is advanced.



## Advanced Stuff

- `the index`
- `hook scripts`
- grab-bag of useful things
  - `git add --interactive` and `patch adding`
  - `git stash [--keep-index]` to tuck things away for a second
  - `git cherry-pick` to pluck just a few cool things someone else did out of one of their commits
  - `git fsck [--unreachable]` as the final defense against lost stuff
  - `git reflog` to see the true history
  - `git bisect` to find out when an error was introduced

On to hook scripts.

This will take about thirty seconds, since you all either do some bash scripting or that would have to be its own talk anyway...

# Hook Scripts

```
.git
├── HEAD
├── config
├── description
├── hooks
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── pre-rebase.sample
│   ├── prepare-commit-msg.sample
│   └── update.sample
├── info
│   └── exclude
└── objects
    ├── info
    ├── pack
    └── refs
        ├── heads
        └── tags
```

8 directories, 12 files

when you init a new git repository in the more recent versions of git,

## Hook Scripts

```
.git
├── HEAD
├── config
├── description
└── hooks
    ├── applypatch-msg.sample
    ├── commit-msg.sample
    ├── post-update.sample
    ├── pre-applypatch.sample
    ├── pre-commit.sample
    ├── pre-rebase.sample
    ├── prepare-commit-msg.sample
    └── update.sample
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

8 directories, 12 files

it always creates a few sample hook scripts for you, in the “hooks” directory.

## Hook Scripts

.git/hooks/pre-commit.sample :

```
#!/bin/sh
#
# An example hook script to verify what is about to be committed.
# Called by "git commit" with no arguments. The hook should
# exit with non-zero status after issuing an appropriate message if
# it wants to stop the commit.
#
# To enable this hook, rename this file to "pre-commit".
#
if git rev-parse --verify HEAD >/dev/null 2>&1
then
    against=HEAD
else
    # Initial commit: diff against an empty tree object
    against=4b825dc642cb6eb9a060e54bf8d69288fbee4904
fi
#
# If you want to allow non-ascii filenames set this variable to true.
allownonascii=$(git config hooks.allownonascii)
...
...
```

The samples tell you generally what to do, but note that there isn't a sample for every event that can trigger a hook.  
you'll have to look online to find them all  
this one verifies that any filenames you're trying to commit contain only ascii characters.  
another useful thing to do might be to bootstrap your rails database on a git checkout to ensure consistency, etc.

## Advanced Stuff

- `the index`
- `hook scripts`
- grab-bag of useful things
  - `git add --interactive` and `patch adding`
  - `git stash [--keep-index]` to tuck things away for a second
  - `git cherry-pick` to pluck just a few cool things someone else did out of one of their commits
  - `git fsck [--unreachable]` as the final defense against lost stuff
  - `git reflog` to see the true history
  - `git bisect` to find out when an error was introduced

last thing before we move to troubleshooting: useful things you'll need occasionally  
the goal here is mostly to make you *aware* of these commands, so you can look them up when you need them

## git add -i

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   /src/main/java/gov/ic/dodiis/i2ps/utils/MongoHelper.java
```

### interactive git-add

often what you've changed would be awful to type, so you select it with your mouse, copy it, and paste it back in.

## git add -i

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   /src/main/java/gov/ic/dodiis/i2ps/utils/MongoHelper.java
```

```
$> git add -i
      staged  unstaged path
      1: unchanged +1/-0 /src/main/java/gov/ic/dodiis/i2ps/utils/MongoHelper.java

*** Commands ***
 1: status      2: update      3: revert      4: add untracked
 5: patch      6: diff        7: quit        8: help
What now> u
```

there is an interactive add, which lets you pick the files by number and even let's you add *individual* changes inside a file, instead of the whole file at once  
we'll enter u (or 2) to “update” the index

```
git add -i
```

	staged	unstaged	path
1:	unchanged	+1/-0	/src/main/java/gov/ic/dodiis/i2ps/utils/MongoHelper.java
Update>>	1		

it shows us the list of unstaged changes and we'll select the item labeled 1

```
git add -i
```

	staged	unstaged	path
*	1:	unchanged	+1/-0 /src/main/java/gov/ic/dodiis/i2ps/utils/MongoHelper.java
Update>> [REDACTED]			

notice there's a little asterisk to the left now to show us we've staged it

## git add -i

```
updated one path  
*** Commands ***  
1: status      2: update     3: revert      4: add untracked  
5: patch       6: diff        7: quit        8: help  
What now> s
```

it tells us we just updated one path, and we'll ask it for the status with "S"

## git add -i

```
staged  unstaged  path
1: +1/-0  nothing  /src/main/java/gov/ic/dodiis/i2ps/utils/MongoHelper.java

*** Commands ***
1: status      2: update    3: revert      4: add untracked
5: patch      6: diff       7: quit        8: help
What now> q
```

```
$> git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>" to unstage)
#
#   modified:  /src/main/java/gov/ic/dodiis/i2ps/utils/MongoHelper.java
```

there's the updated status again, and we'll do "q" to quit  
and when we run regular git status, now it's listed as a change to be committed

## git add -i

```
      staged  unstaged  path
1: +1/-0    nothing   /src/main/java/gov/ic/dodiis/i2ps/utils/MongoHelper.java

*** Commands ***
1: status      2: update     3: revert      4: add untracked
5: patch      6: diff       7: quit        8: help
What now> q
```

```
$> git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   /src/main/java/gov/ic/dodiis/i2ps/utils/MongoHelper.java
```

interactive add is cooler than this, and I encourage you to try this one out on a file where you've made multiple changes.

it allows you to add individual patches in the file to the index to be committed, instead of the whole file

- useful for making neat, conceptually-grouped commits

## git stash

- git stash
- git stash list
- git stash pop
- git stash apply

these will go faster now

git stash tucks all of your uncommitted working directory work away for just a bit, most commonly so you can change branches without messing up your uncommitted work,

and you can bring it back later with git stash pop (which removes the stash from the stash list) or git stash apply, which leaves it there

view it with list

```
git stash --keep-index
```

something very cool about git stash is the --keep-index option, which will stash all changes *except for those you've already staged with git add*  
this is very useful in conjunction with patch-adding, for testing to see if a patch-added commit actually builds

```
git cherry-pick <commit>
```

default behavior of git cherry-pick applies the changes introduced in a particular commit and records a new commit for them  
very useful for pulling in just a few of the changes in someone else's hot idea branch after a git fetch

## git fsck [--unreachable]

git fssssck is the final defense against lost things

I won't elaborate, but if you've ever committed or stashed something and then seemed to lose it in any way, it is almost certainly still retrievable, but you might need this command to find it. When you do, make a new branch that points to it so you don't lose it again

```
.git  
├── COMMIT_EDITMSG  
├── HEAD  
├── config  
├── description  
├── hooks  
│   └── ...  
├── index  
├── info  
│   └── exclude  
└── logs  
    ├── HEAD  
    └── refs  
        └── heads  
            ├── master  
            └── new_branch
```

## git reflog

```
$> git reflog  
cc9fc8c HEAD@{0}: checkout: moving from  
master to new_branch  
cc9fc8c HEAD@{1}: commit (initial): init
```

```
$> git reflog master  
cc9fc8c master@{0}: commit (initial): init
```

```
$> git reflog new_branch  
cc9fc8c new_branch@{0}: branch: Created  
from master
```

15 directories, 24 files

the reflog tracks changes to where the references are pointing, and anything that happened while they were pointing there

it is your “true history”

useful for figuring out how things originally were before a rebase or other history rewrite

## git bisect

- git bisect start
- git bisect bad
- git bisect good [<commit>]
- git bisect reset

git bisect will help you find where an error was introduced into your code, and thus probably what changes caused that error, by doing binary search through your history

## git bisect

```
$> git bisect start  
$> git bisect bad  
$> git bisect good v1.0  
Bisecting: 6 revisions left to test after this  
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

you start with `git bisect start`,  
then you tell bisect that the current state is bad, and that the last known good state  
was the commit pointed to by this tag here  
`git automatically checks out a commit in between those`

## git bisect

```
$> git bisect start  
$> git bisect bad  
$> git bisect good v1.0  
Bisecting: 6 revisions left to test after this  
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo  
  
$> git bisect good  
Bisecting: 3 revisions left to test after this  
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

there's nothing wrong here, so you tell git and it checks out another commit, half-way between the one you just tested and your bad commit

## git bisect

```
$> git bisect start
$> git bisect bad
$> git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo

$> git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing

$> git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

this one *is* bad, so you tell git with git bisect bad

## git bisect

```
$> git bisect good  
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit  
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04  
Author: PJ Hyett <pjhyett@example.com>  
Date: Tue Jan 27 14:48:32 2009 -0800  
  
    secure this thing  
  
:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730  
f24d3c6ebcf639b1a3814550e62d60b8e68a8e4 M config
```

and this one is good, so you tell git, and it now has enough information to know which commit introduced the error  
and it shows information about that commit

## git bisect

```
$> git bisect good  
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit  
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04  
Author: PJ Hyett <pjhyett@example.com>  
Date: Tue Jan 27 14:48:32 2009 -0800  
  
    secure this thing  
  
:040000 040000 40ee3e7821b895e52c1695092db9bcd4c61d1730  
f24d3c6ebcf639b1a3814550e62d60b8e68a8e4 M config
```

## git bisect reset

finally, run `git bisect reset`, which will return your HEAD to where it was before you started  
or you'll get yourself into a weird state  
you can also give `git bisect` a script to run that determines if the code is good or bad,  
and it'll do everything automagically for you

## Understanding Git

- ~~The Basics~~
- ~~The Git Object Database~~
- ~~History Inspection~~
- ~~Branching Workflows~~
- ~~Collaboration~~
- ~~Advanced Stuff~~
- Troubleshooting

Oh good, we're almost half-way through the talk.

# Troubleshooting

*troubleshooting* is something best learned while you're in the middle of a problem, so I'll do the same thing I did with the grab-bag of useful commands, and just make you aware of some things,  
and you can check these slides later when you need them

## Troubleshooting

changing your mind about what you want to commit

- `git reset`
- `git checkout HEAD filename`
- `git reset --hard`

how to change your mind about what you want to commit

`git reset` with no arguments is the opposite of `git add`,

`git checkout HEAD filename` will permanently nuke the changes made to that file,

and `git reset --hard` will permanently nuke all changes since the last commit

the last two are dangerous, since they destroy uncommitted changes!

## Troubleshooting

changing your mind about a commit after you've already committed

- git commit --amend

how to change your mind about a commit after you've already committed  
if you just committed something and realized you didn't add all your files, or you messed up the commit message,

git add whatever you want and run git commit --amend  
this is actually short-hand for rebasing onto HEAD~ and then committing - it undoes the last commit, turns it into a patch, and commits afresh

## Troubleshooting

ditch a commit by moving the branch pointer

- `git reset --hard <target_commit>`
- `git branch -f <branch_name> <target_commit>`

how to ditch a commit by moving the branch pointer back  
you can ditch a commit or just move around and explore by moving the branch  
pointer before continuing work  
with `git reset --hard` or `git branch -f` (for “force”)

## Troubleshooting

uncommit things you didn't mean to commit yet  
(or ever)

- git revert - makes a “go back” commit
- just move the branch back to an earlier commit
- the nuclear option: git filter-branch

how to uncommitt things you didn't mean to commit  
there are three options -

- 1) reversion, which is always an option,
- 2) just ditch the commit by moving back to an earlier commit before continuing work,
- 3) and the nuclear option, which you'd need for accidentally-committed sensitive data or enormous files.

git filter-branch can actually go back through your entire history and *permanently* remove a file, rewriting every commit since you committed that file.

The last two will wreak havoc on anyone who's already based work off of yours

## Troubleshooting

how to avoid these problems in the first place:

.gitignore

and a good way to avoid these problems in the first place is a good .gitignore file before you start working

## Troubleshooting .gitignore

```
# ~/.gitconfig file: [core]
    excludesfile = /Users/your.name/.gitignore
```

(then create a .gitignore in your home dir.

you can actually have a global .gitignore file if you edit your configuration like this  
and then make a .gitignore file in your home directory

## Troubleshooting .gitignore

```
# ~/.gitconfig file: [core]
    excludesfile = /Users/your.name/.gitignore
```

(then create a .gitignore in your home dir.

Here's what's in mine:

```
.DS_Store
*.svn
*~
.*.swp
.powenv
.classpath
.project
.settings
target
*.class
```

[Check this out for more information](#)

here's what's in mine

## Troubleshooting

problems arising from files being yanked out  
from underneath the feet of open editors

this is just something to be aware of - some editors won't notice when a git checkout,  
merge or pull, for example, yanks the files out from under it and replaces them with  
new ones

you could accidentally overwrite someone else's work

## Troubleshooting

- changing your mind about what you want to commit
  - git reset, git checkout, git reset --hard
- changing your mind about a commit after you've already committed
  - git commit --amend
- ditch a commit by moving the branch pointer back
  - git reset --hard ... or git branch -f ...
- how to uncommit things you didn't mean to commit yet
  - git revert
  - just move the branch back to an earlier commit
  - nuclear option: filter-branch
- how to avoid these problems in the first place
  - .gitignore
- problems arising from files being yanked out from underneath the feet of open editors

and here's the summary

other than that, I just hope this talk has given you enough insight into git that you can hash through any issues that arise

# Review

we're almost done, so a quick review

- git init
- git clone
- git add
- git commit
- git branch
- git checkout
- git merge
- git remote
- git fetch
- git push
- git diff
- git log

**12**  
+ git status

here are the twelve most used git commands, and git status which *my fingers* automatically insert after every command  
git init conjures a new repo from the air,  
git clone makes a local clone of a remote repo,  
git add stages changes to be committed,  
git commit carves them in stone,  
git branch manages branch pointers in the tree of commits,  
git checkout jumps around the branch tips and checks out the relevant files,  
git merge will merge two or more branches, recording a new, multiple-parent commit,  
git remote manages remote repos,  
git fetch downloads new commits in those repos,  
git push *uploads* new commits *to* those repos,  
git diff shows the differences between treeishes...  
and git log shows you the history of commits in your repository.  
git status shows the status.

## .git, One Last Time

```
.git
└── COMMIT_EDITMSG
└── HEAD
└── config
└── description
└── hooks
    └── ...
└── index
└── info
    └── exclude
└── logs
    └── ...
└── objects
    ├── 54
    │   └── 3b9bebdc6bd5c4b22136034a95dd097a57d3dd
    ├── c7
    │   └── dcf9ef54b124542e958c62866d8724471d77d2
    ├── cc
    │   └── 9fc8c4ea4df4f245103cbe80c35bfa2eb07e52
    ├── e6
    │   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
    └── info
        └── pack
└── refs
    ├── heads
    │   ├── master
    │   └── new_branch
    └── tags
        └── taggy
```

15 directories, 24 files

And one last time, the .git directory.  
you should recognize most of what's in here now, and what you don't know about  
yet is either peripheral or easily discovered on git-scm.com

## Resources

- `git help [command]`
  - surprisingly good manual pages, which are also online
- [git-scm.com/docs](http://git-scm.com/docs) and [git-scm.com/book](http://git-scm.com/book)
  - truly beautiful, clear and illustrated documentation, and has a free book that can answer all of your questions
- [GitX](#)
  - great tool for visualization of branching problems, and for designers who would rather not interact with the command line
- [the Getting Git video](#) on Vimeo
  - the video that made git click for me, the source for many of my slides, best hour ever spent
- Casey's cheat-sheet
  - an attachment to the "Git Development Workflow" Six3 Confluence page
- Blog Post: "[A Successful Git Branching Model](#)"
  - excellent blog post on good workflow conventions Casey linked to in the comments there

speaking of which.  
here are some excellent resources

*The End*

Questions?