Intermediate Git

Christian Moscardi

ERD Business Development Staff

2-25-2021

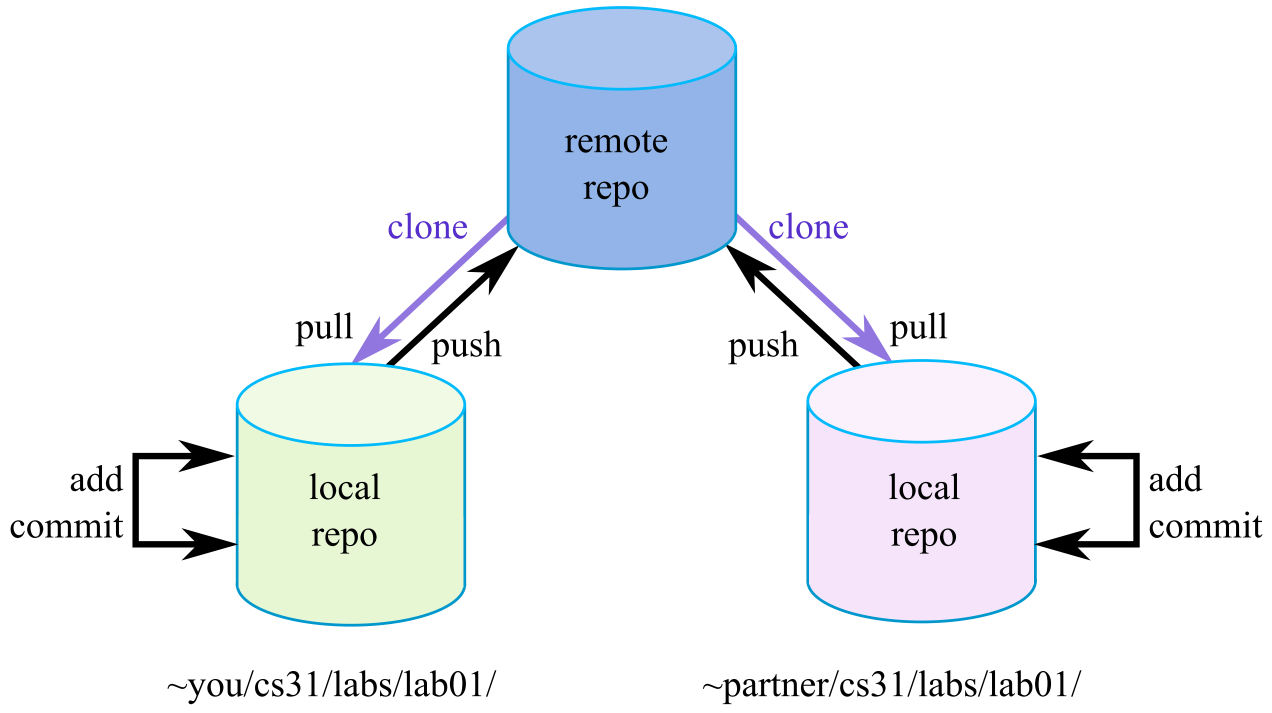
(credit to Zack Newman for original talk)

Expectations for this talk

I assume some familiarity with basic Git operations.

But please stop me if I say something that doesn't make sense.

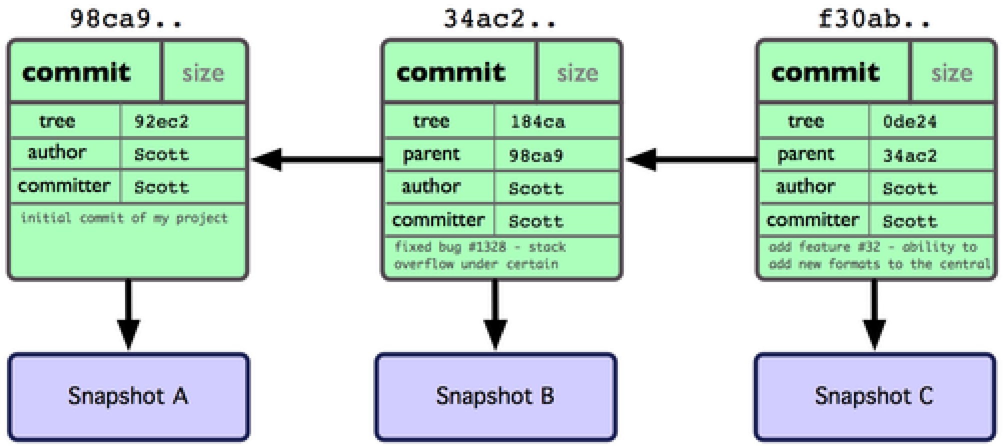
1000-foot overview



(credit to https://www.cs.swarthmore.edu/~adanner/help/git)

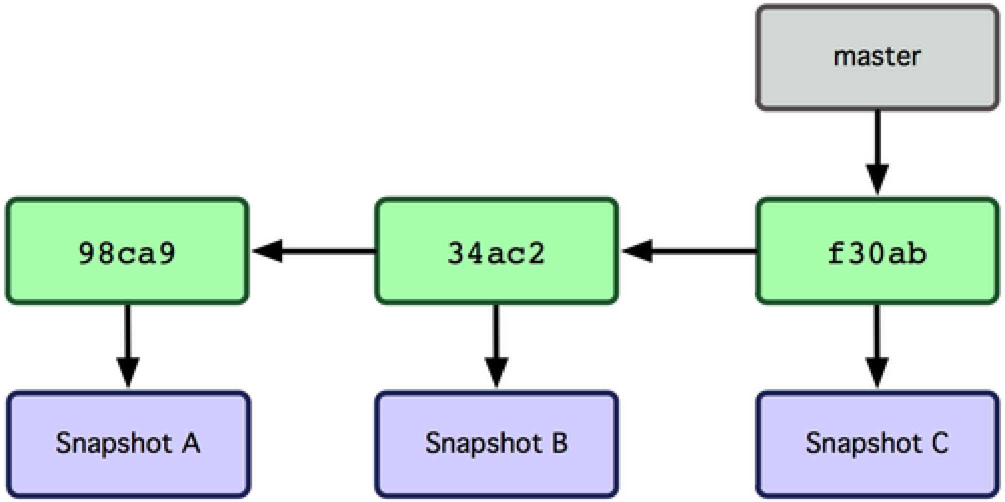
How Git Works

Commits are snapshots of the state of a repo.



Branches

What's a *branch*? Simply a pointer to a commit!

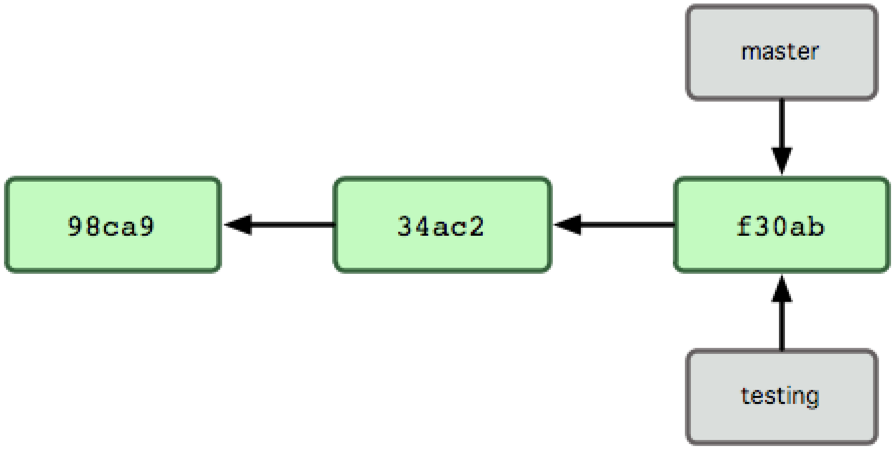


Don't believe me? Check!

```
.no-highlight
> cat .git/refs/heads/master
d12a3c549eb486c5ad5f95c145df4685097755ad
> git show d12a3c54
commit d12a3c549eb486c5ad5f95c145df4685097755ad
Author: Zack Newman <znewman01@gmail.com>
...
```

Branches

```
.no-highlight
git branch testing # create a new branch
```

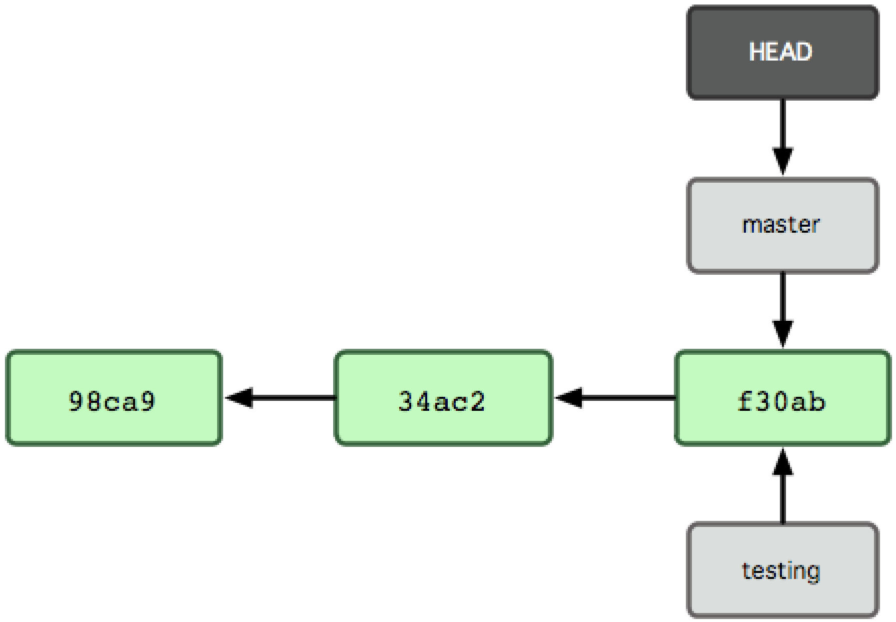


```
.no-highlight
> cat .git/refs/heads/testing
8e19ecbd2b962a7eb4a2aa36b5bac51442a1b2a8
> git show 8e19ecb
commit 8e19ecbd2b962a7eb4a2aa36b5bac51442a1b2a8
Author: Zack Newman <znewman01@gmail.com>
...
```

Branches

How does Git know what branch I'm on?

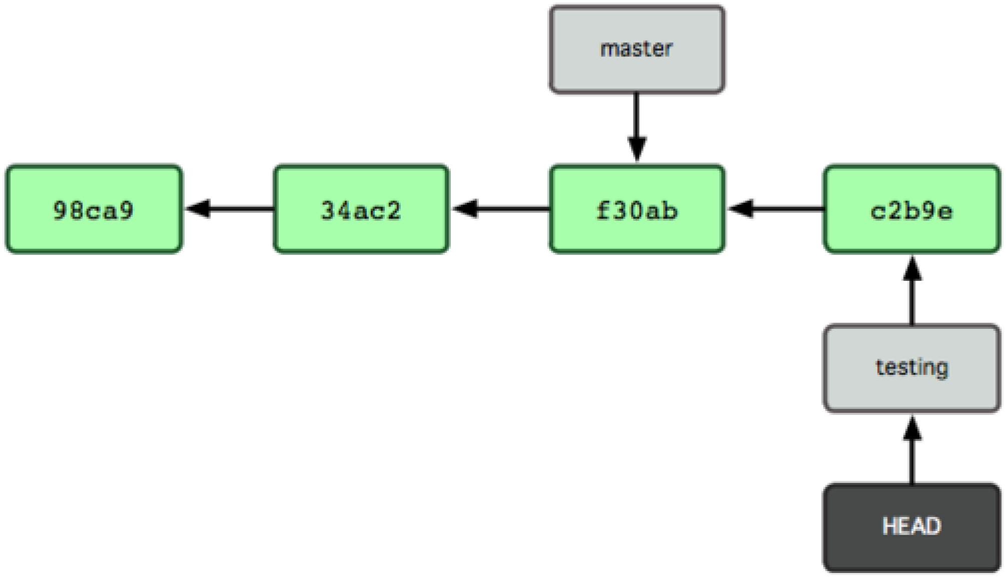
```
.no-highlight
> cat .git/HEAD
ref: refs/heads/master
```



Branches

What happens when I commit?

```
.bash
git checkout testing
# edit edit
git commit
```

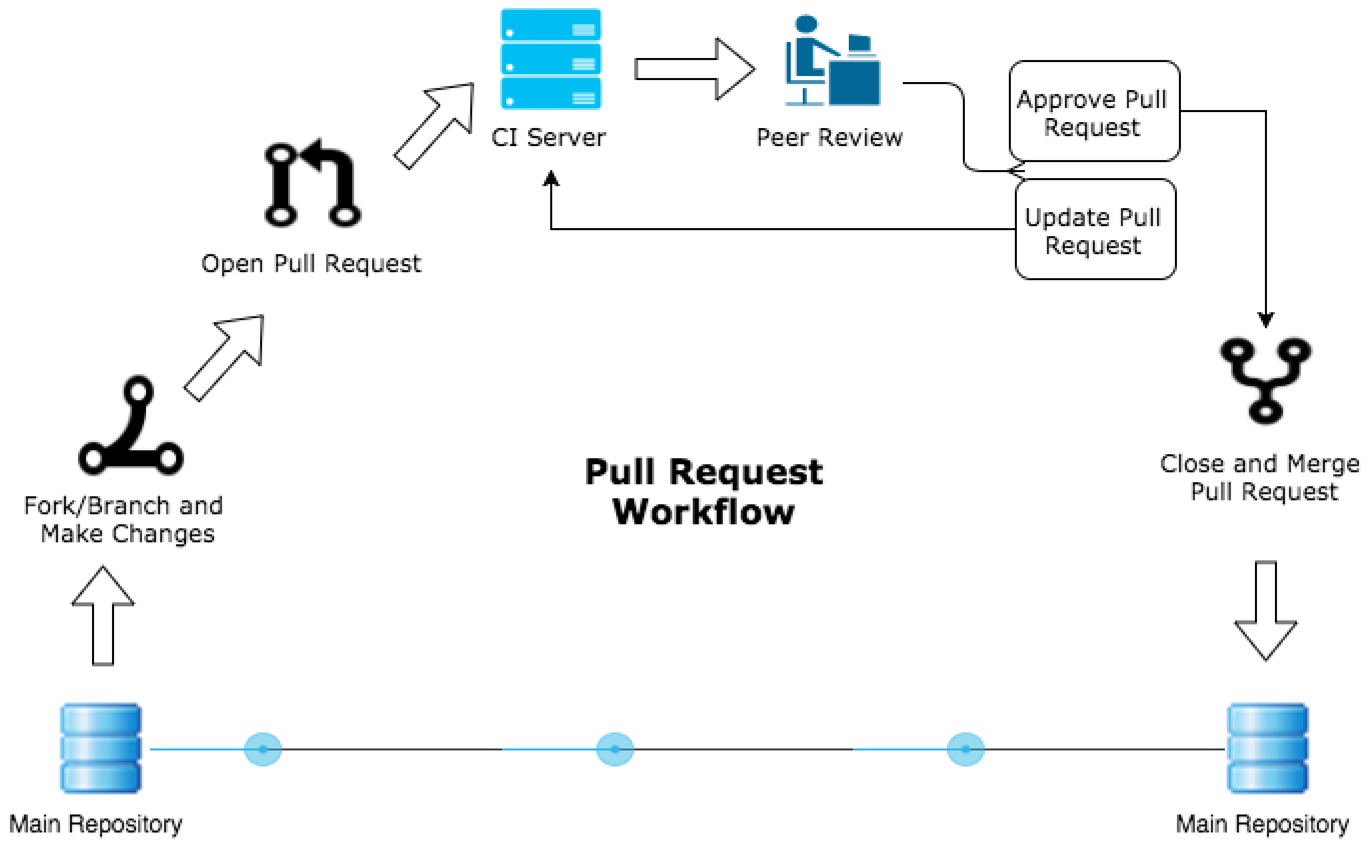


If we were to merge `testing` into `master` right now, it'd be simple (fast-forward).

Branches and remotes

Now, let's zoom back out. We have made some changes that we want to incorporate into production. Using gitlab (or github), what's the workflow?

Merge (or pull) requests!



What's a remote?

A **remote** is another copy of a repository that lives outside of where you're currently working. Typically, this is a server where you're collaborating with your team.

```
.bash
> git remote -v
github  https://github.com/census-bds/intermediate-git.git (fetch)
github  https://github.com/census-bds/intermediate-git.git (push)
origin  https://git.econ.census.gov/mosca303/intermediate-git-demo.git (fetch)
origin  https://git.econ.census.gov/mosca303/intermediate-git-demo.git (push)
```

How do I make changes to a remote?

In a typical workflow, you develop new functionality on a branch, then **push** that branch to the remote where you're collaborating.

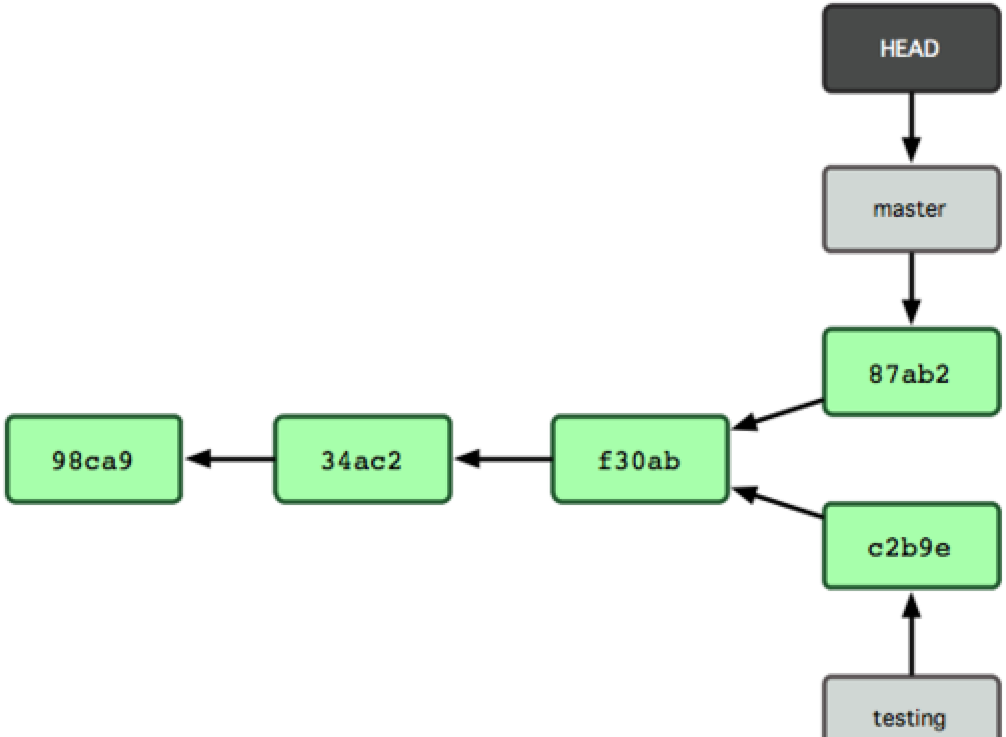
```
.bash
git push origin [branch_name]
```

Git pushes all of the necessary commits and branch metadata to the remote server.

Branches

Let's make things more interesting:

```
.bash
git checkout master
# edit edit
git commit
git push origin master
```



Merging

Git will:

1. Find common ancestor
2. Identify commit to merge into and commit to merge in
3. Create a new commit based on three-way merging. This commit has two parents.

If this is successful, it is safe to delete the branch.

```
.bash
git branch -d testing
```

Local command

```
.bash
git merge testing
```

Remote command

```
.bash
press the merge button on 'testing' merge request
```

Merge Conflicts

A lot of times, merging *won't* be successful.

Good news: This isn't actually *that* painful of a process. Git provides some tools to make it easier.

Bad news: You're still going to have to do it *a lot*.

Merge Conflicts

Merge conflicts occur when Git can't figure out how to apply two diffs to the same blob.

`app.py -- HEAD`

```
import math
```

`app.py -- master`

```
from math import sqrt
```

`app.py -- new_branch`

```
import math

print(math.sqrt(2))
```

Merge Conflicts

```
.no-highlight
Auto-merging app.py
CONFLICT (content): Merge conflict in app.py
Automatic merge failed; fix conflicts and then commit the result.
```

See what's up using `git status` (not shown).

`app.py`

```
from math import sqrt
=======
import math

print(math.sqrt(2))
>>>>>>> testing
```

We **want** there to be a merge conflict in this case so that we can resolve it properly:

```
from math import sqrt

print(sqrt(2))
```

Then,

Merge Conflicts

That's really all there is to it. Some tips/tricks though:

- `git mergetool`
- `git checkout --theirs|--ours [file]`

Rewriting History

Make sure **never** to do any of the following things after you've pushed your changes to a remote repo.

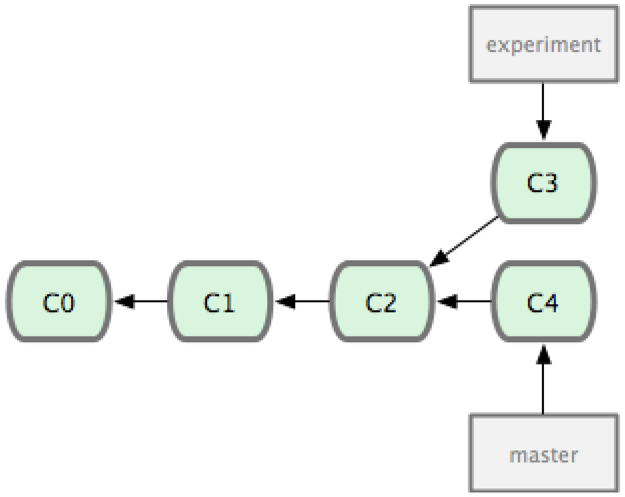
Rewriting History

Forgot a file? Messed up the commit message?

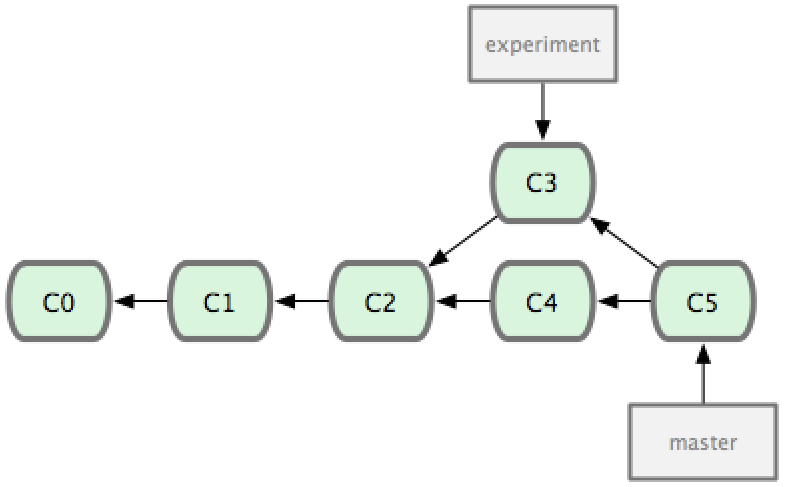
```
.bash
git commit --amend
```

Rewriting History

Alternative to merging: the rebase



$\rightarrow$

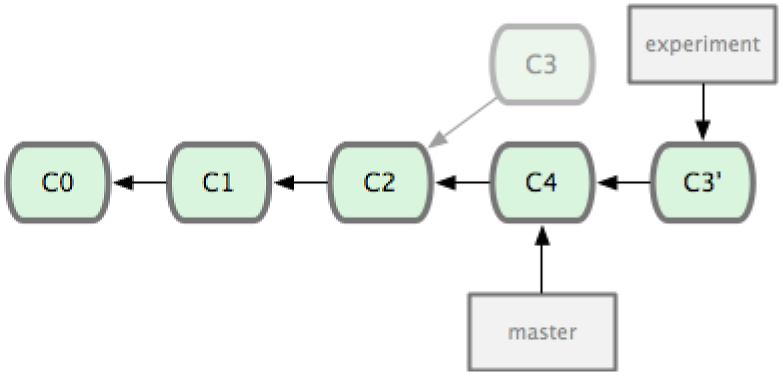


Rewriting History

```
git checkout experiment
git rebase master
```

This will:

1. Find the common ancestor of `experiment` and `master`.
2. Getting all of the diffs of the commits from `experiment` and saving them to temporary files.
3. Setting `experiment` to point to the same commit as `master`.
4. Apply each of the diffs in sequence.



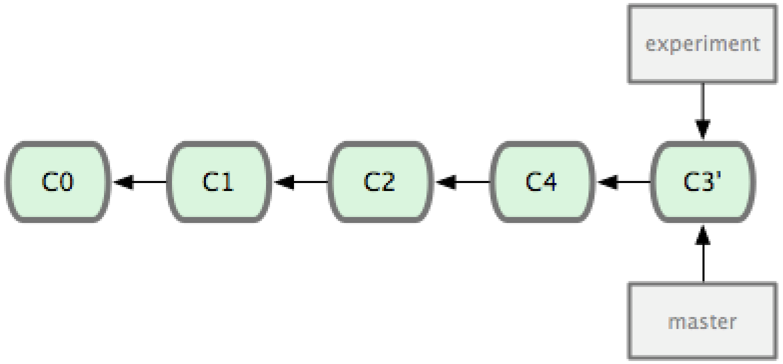
Merge conflicts can occur here.

Rewriting History

At this point, you can simply

```
.bash
git checkout master
git merge experiment
```

and `master` gets fast-forwarded.



This is sweet because you don't get any ugly merge-fix commits in `master`.

Rewriting History

You can also interactive rebase. Say you've been hacking on a feature branch and it's really ugly, but you want to clean it up before you merge into `master`.

```
.bash
git checkout -b feature
# hack hack
git commit -a
# hack hack
git commit -a
# etc.
git rebase -i master
```

Rewriting History

This will put you into interactive rebase mode:

```
.no-highlight
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

Note that this runs old-to-new. You can do any of the commands, then exit. Git will give you pretty clear instructions on what to do from there.

Tagging

If you're maintianing a library or something that gets actually released, tags are really helpful.

At its simplest, a tag is essentially a branch that doesn't change: it points to the same commit forever.

Tags can also be annotated with messages, GPG-signed, and have metadata.

```
.bash
git tag v1.0 # create plain tag
git tag -a v1.0 # create annotated tag
git tag -s v1.0 # create signed tag
git tag -v v1.0 # verify signed tag
git push origin v1.0 # push tag to remote, which doesn't happen by default
```

Please please please use semantic versioning.

Configuration

`git config` takes:

- `--system`: `/etc/gitconfig`
- `--global`: `~/.gitconfig`
- default: `.git/config`

Some useful `git config` commands:

```
.bash
git config --help

# you should definitely do these two
git config --global user.name "Zack Newman"
git config --global user.email znewman01@gmail.com

# do this one!
git config --global color.ui true

# a little more controversial
git config --global pull.rebase true

# aliases are fun, but I won't dive too deep here
git config --global alias.co checkout
git config --global alias.hi !echo hi
```

Configuration

`.gitignore` files are really useful!

You should have a global gitignore. Mine lives at `~/.gitignore`. Set that location:

```
git config --global core.excludesfile ~/.gitignore
```

Example contents:

```
.no-highlight
*~ # editor backups!
.DS_Store # if you use a Mac and commit one of these files I will be very upset
venv # anything specific to YOUR system goes in YOUR gitignore
```

Best Practices

- Don't commit clutter!
- Don't commit data!
- Each commit should stand on its own.
- Don't combine two different changes into one commit (whitespace commits should be separated).
- Don't leave empty space at the end of lines (`git diff --check`).
- Don't commit lines that are longer than your project's convention (I like 80 chars).
- Don't commit commented out code.

Best Practices

```
.no-highight
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug"
or "Fixes bug."  This convention matches up with commit messages generated
by commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded by a
  single space, with blank lines in between, but conventions vary here

  - Use a hanging indent
```

(Credit to Tim Pope)

Misc. Tips and Tricks

`git commit` options:

- `-a`: add all files currently in the repo to the staging area before committing.
- '-m': specify commit message on the command line. Don't do this in a real project; write a good commit message.
- '-v': *verbose*: include the diff for this commit in the editor for the commit message so you can remember what you did.
- '-p': commit interactively by chunks. This is crazy: you can commit some parts of files but not others.

`git mv`: git detects moved files automagically

Bash autocompletions: Google and install, it'll make your life easier. Or use zsh.

Misc. Tips and Tricks

`git stash`: store files from your working directory without committing them.

```
# example usage
git checkout -b feature_branch
# hack hack hack
# your boss says there's a critical bug in production
git stash
git checkout master
# fix fix fix
git commit
git push
git checkout feature_branch
# you may also want to rebase feature_branch onto master here
git stash pop
```

`git reflog`: tracks every step you've taken locally

```
# example usage
git mergetool
git commit

# oh crap, messed that up
git reflog
# find hash of commit before resolving conflict

# WARNING: git reset itself is dangerous
# never do anything like this with changes you've pushed
```

The Future

Further resources:

- christian.l.moscardi@census.gov
- *Pro Git* by Scott Chacon
- StackOverflow (seriously)
- Git does have good `man` pages

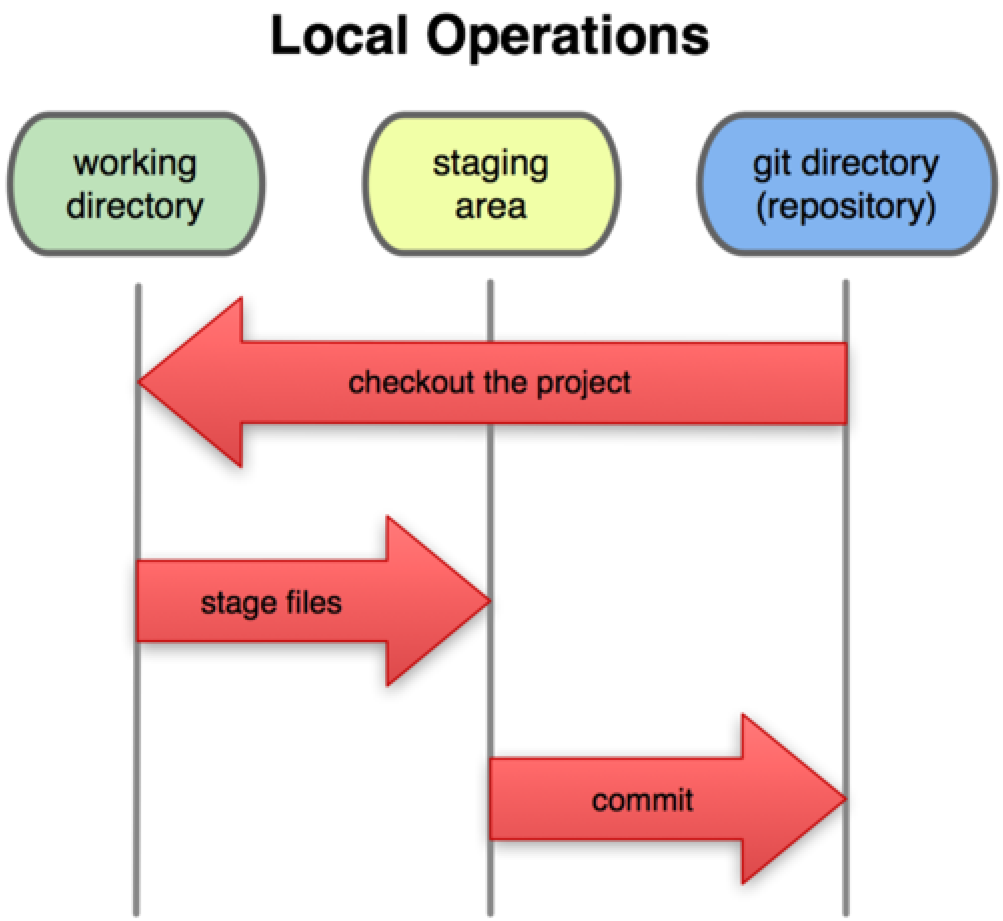
Other topics:

- Recovery
- `git bisect`
- git hooks

How Git Works

Files are *commited*, *modified*, or *staged*

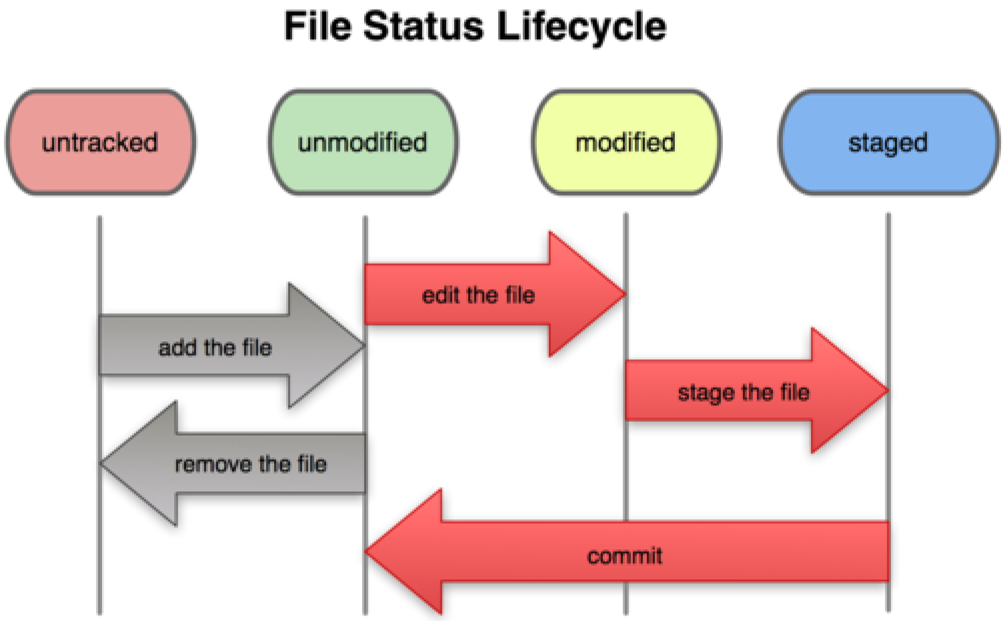
Stored in the Git directory, working directory, or staging area respectively



How Git Works

What is a commit?

- Snapshot of the state of your repo
- **Not** a diff (unlike in Subversion)


File Status Lifecycle
untracked
unmodified
modified
staged
edit the file
add the file
stage the file
remove the file
commit