Everything you want to know about

Git

A TECH TALK BY BEN CHAPMAN-KISH

MARCH 29, 2018

Topics

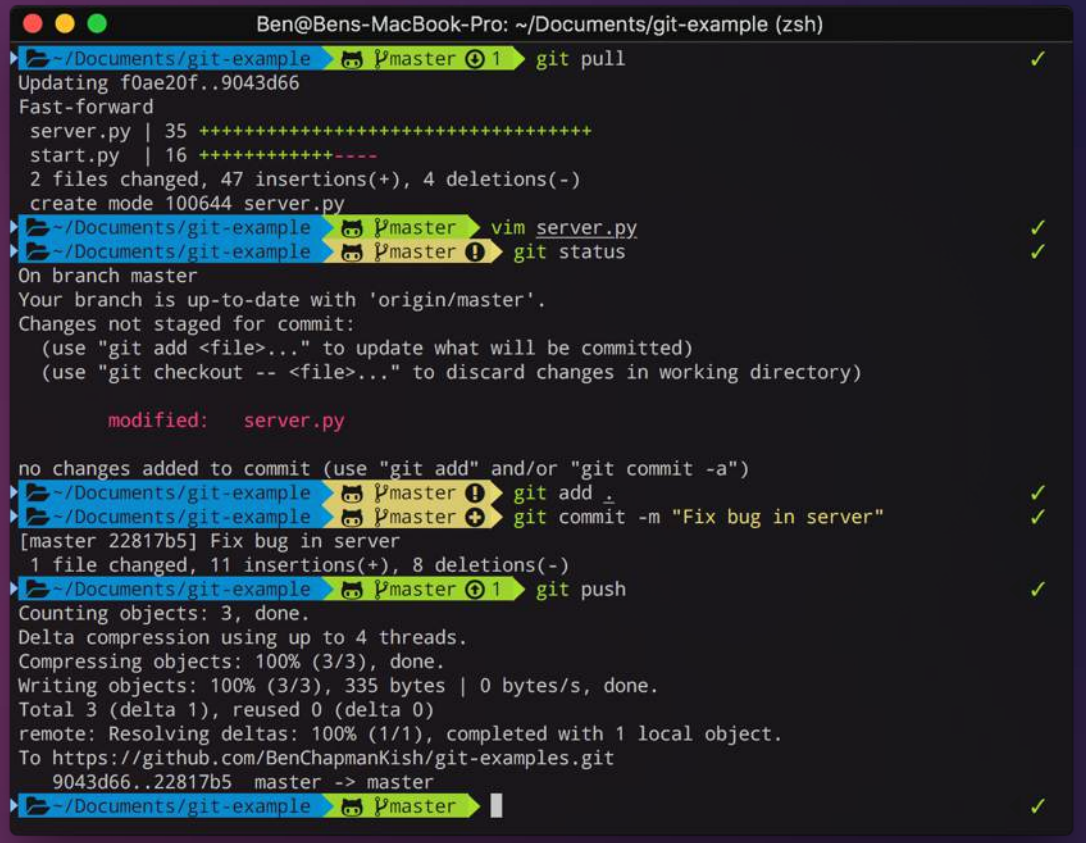
- Intro
- Common Git workflows
- Commits and branches
- More commands and tools
- Tips and tricks
- How Git *really* works

What is Git, again?

- Distributed version control system
- Created by Linus Torvalds in 2005
- Open-source, well-maintained
- High performance, powerful, secure

Everyday use of Git

- Pull
- Push
- Checkout
- Branch
- Merge
- Add
- Commit
- Tag
- Status
- Diff

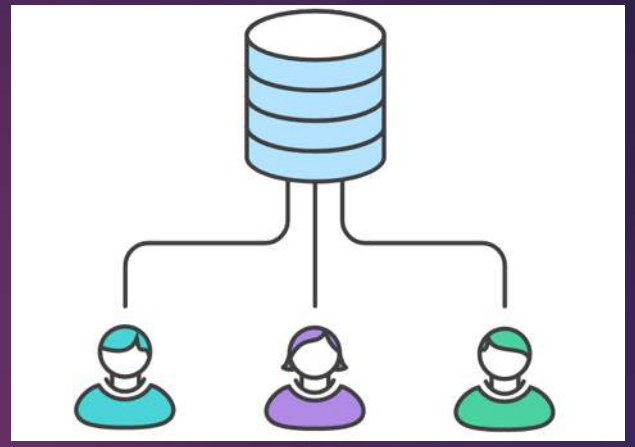


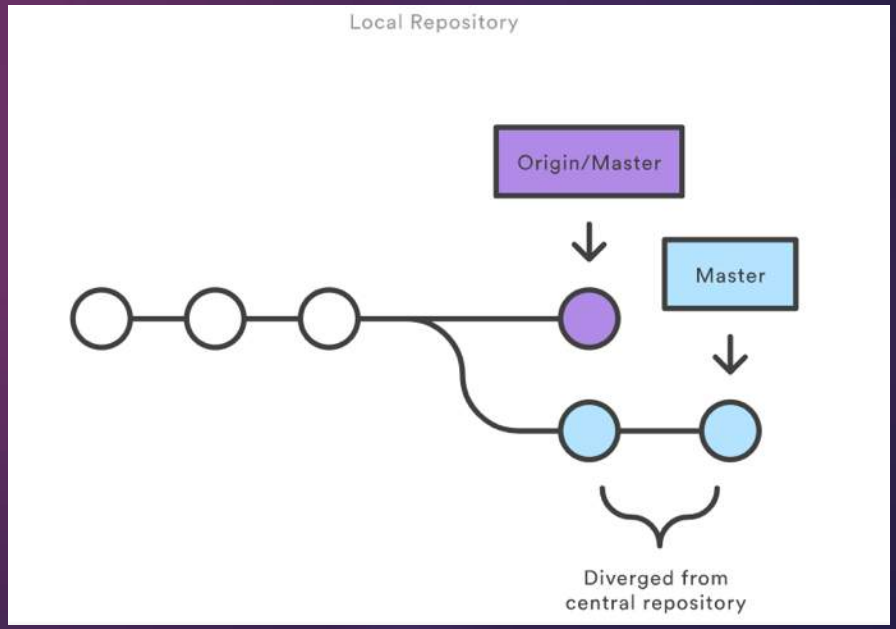
… among many others, depending on your workflow

Common Git Workflows

Basic/Centralized

- Essentially only one branch on one central repository
- Every dev makes their own local changes and pulls/pushes as necessary
- Ideal for small teams or teams transitioning from SVN



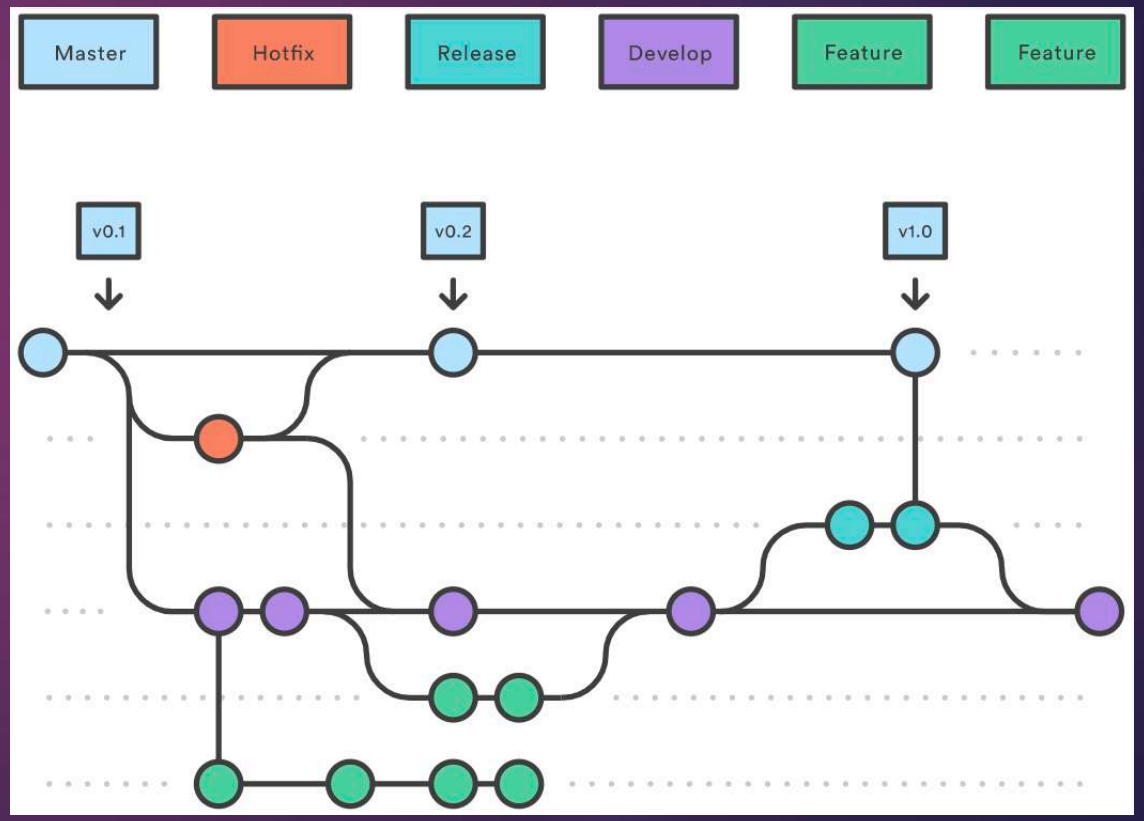


Feature branches

- Single branch with infinite lifetime – master
- Branch off for each feature, then merge/rebase back in when it's done
- Easier to work on concurrent features than centralized workflow

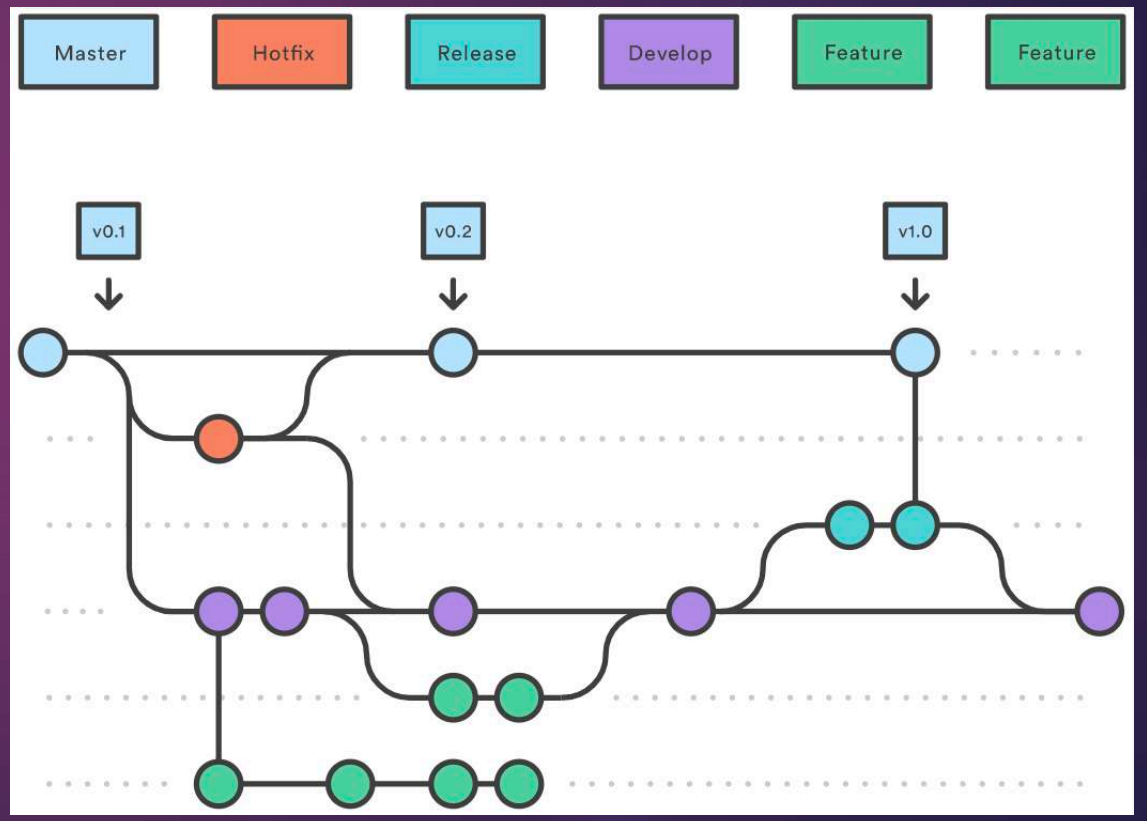
Git Flow

- Two parallel branches with infinite lifetime – master and develop
- Master is production-ready code
- Develop is integration branch for all features
- Feature branches are branched off and merge back into develop



Git Flow

- Release branches branch off develop, only bug fixes are made
- When it's ready, the release branch merges into master and develop
- Hotfix branches branch off and back into master to quickly fix bugs in prod



Git Flow

- Ideal for large or release-based projects
- Makes parallel development very easy
- Offers release staging area for fixes and testing before shipping
- Offers dedicated channel for hotfixes to production

Forking

- Each dev has their own “fork” of the official repo
- Changes are made on a new branch in the forked repo
- When the changes are done, a pull request is made from the new branch to the official repo
- Often used in public open-source projects and in conjunction with hosting services such as GitHub

Before we go any further…

Commits and Branches

What exactly is a commit?

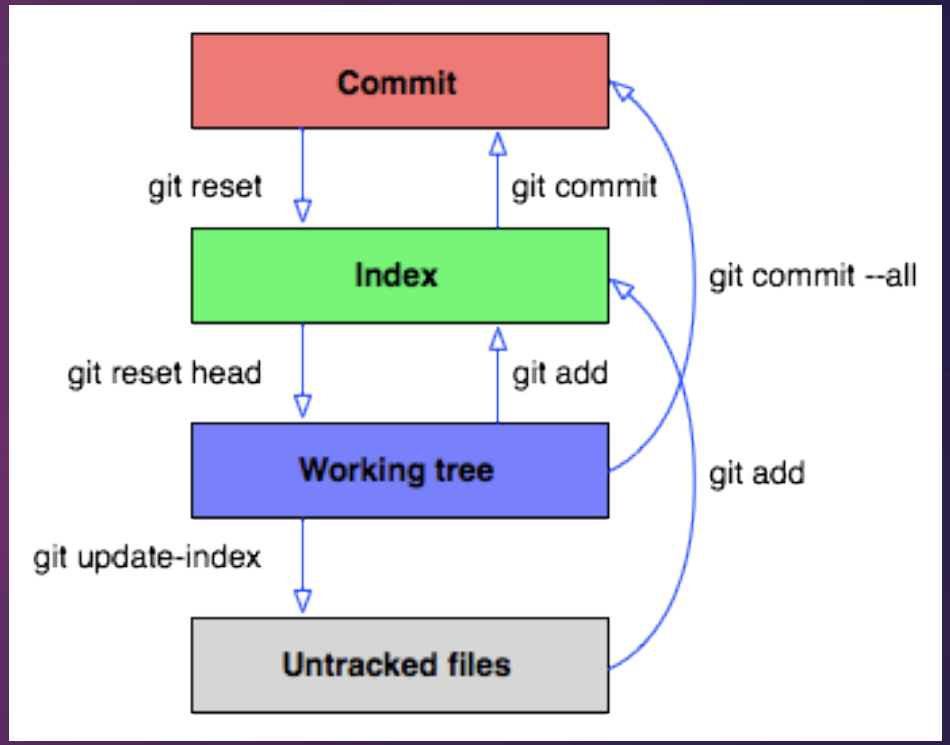
- A commit is a snapshot of the state of the project at a certain point
- Commits also contain metadata, such as:
  - The commit message
  - The author of the commit
  - The time the commit was made
  - Its parent commit(s), if any
- Every commit can be uniquely identified by its SHA-1 hash

What exactly is a commit?

- A commit's hash is generated from the "snapshot" and its metadata
- When you change a commit, you're really making a new commit with new data and a new hash
- If the changed commit has children, every child commit will have to be recreated with their parent hashes updated

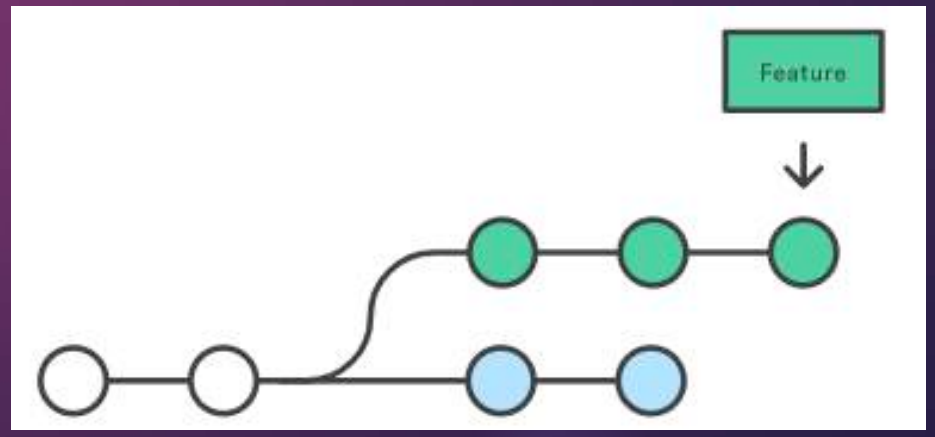
Local Operations

- When you modify files, you're updating your *working tree*
- When you add a file, you're moving it to the *staging area*, where it can be stored in a commit
- When you commit, Git essentially saves the files in the staging area in a commit object



Branches

- All a branch really is is a reference to a certain commit
- When you make a new commit on the a branch, Git automatically updates the branch to point to your new commit
- When you create a new branch, it points to the same commit that the branch you were just on did



Tags

- A tag is a reference to one particular commit
- Unlike branches, which update automatically, a tag will always point to the same commit
- Useful for marking versions

Head

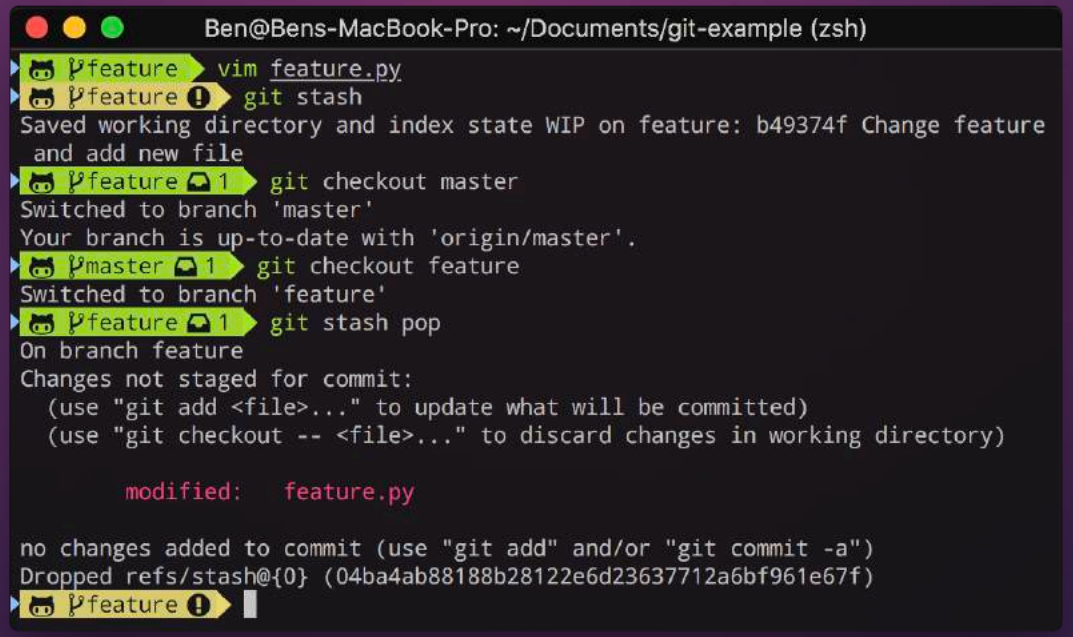
- *Head* is a reference to the latest commit on the currently checked-out branch
- Many operations you do are implicitly on *head* without you knowing
- When you checkout a branch, all that's actually happening is Git changes what *head* points to

What's next?

More Commands and Tools

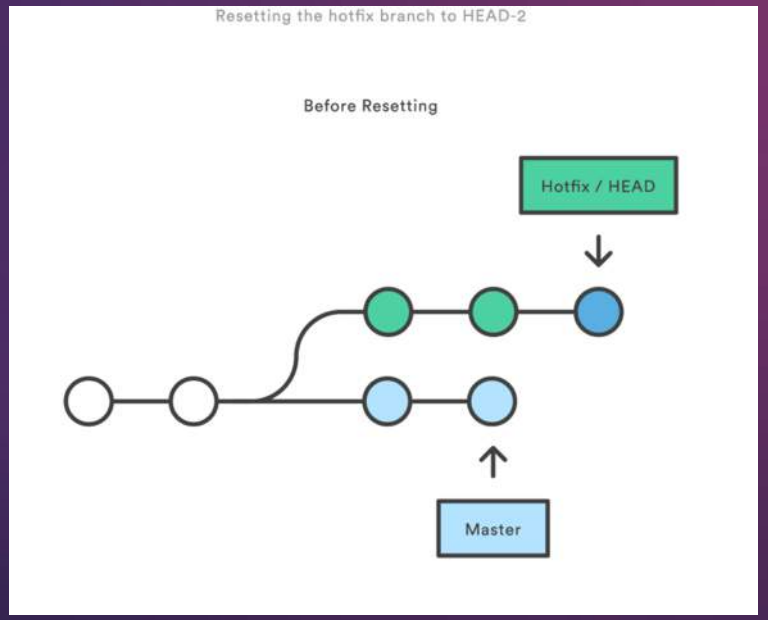
Stash

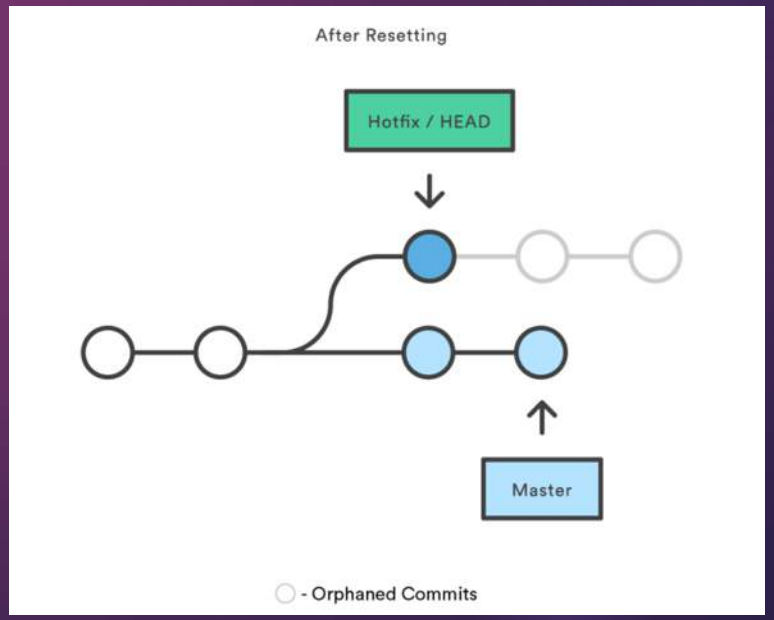
- A "pseudo-commit" stored in a special place
- For work-in-progress changes that aren't ready to be committed, but you need to checkout a different branch



Reset

- Often used to reset changes and go back to an earlier commit, but…
- Fundamentally, *reset* just moves a branch head
- Has options to keep changes in the working tree or staging index, if desired

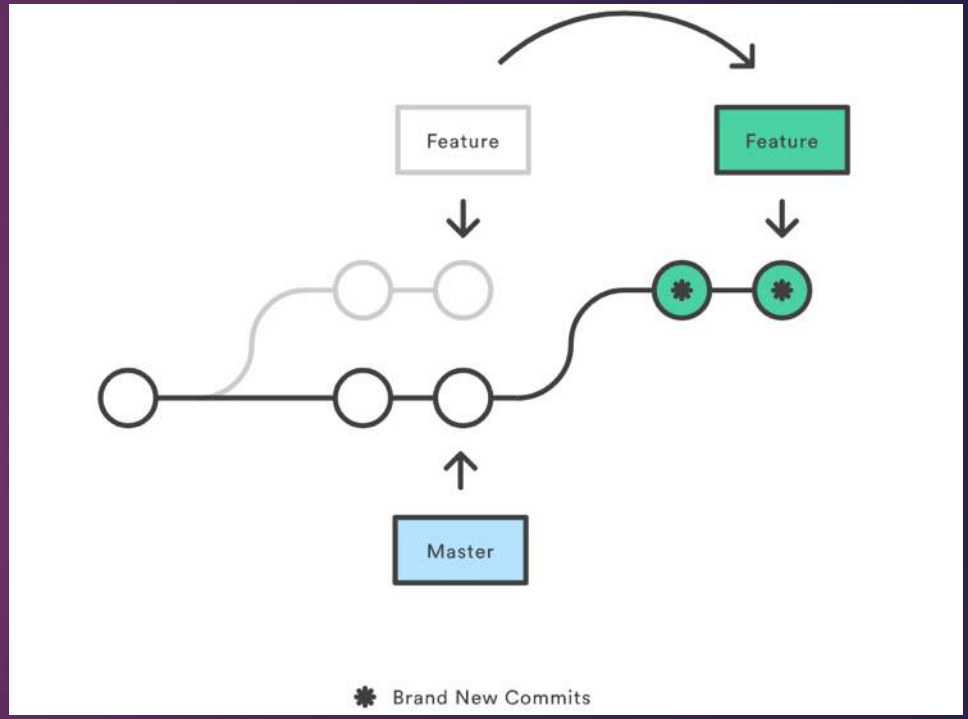




Rebase

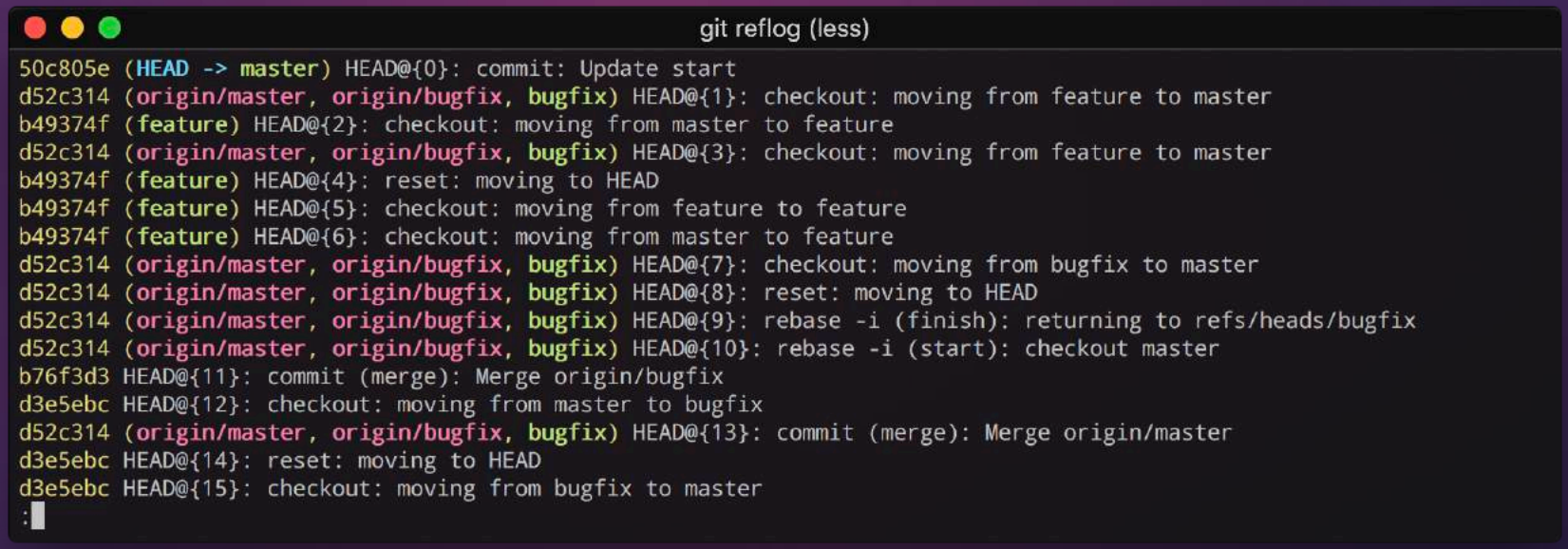
- Reapply commits on a different parent
- Often used to preserve linear history
- Can also be used to drop, squash, and edit any commit

- Note: rebasing *changes history*, don't do this on a shared branch



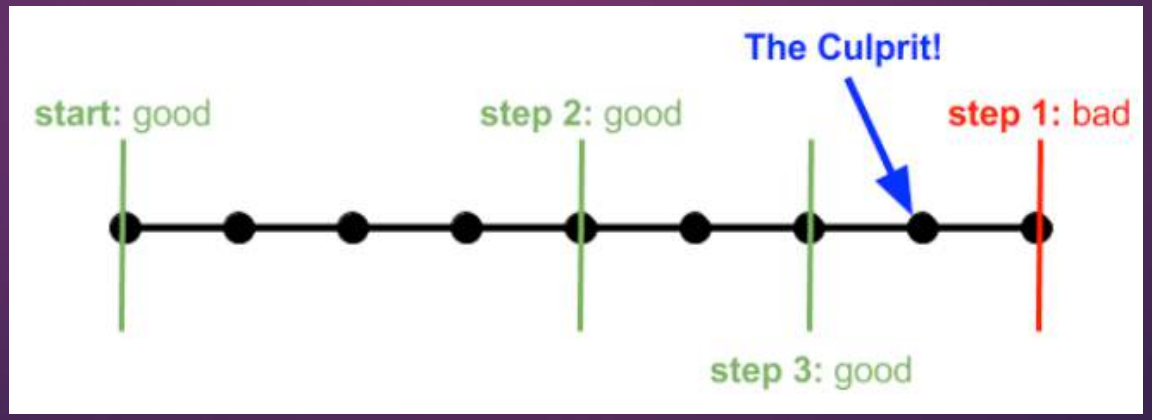
Reflog

- See a history of braches/commits you've checked out
- Especially useful if you accidentally reset too far and lose a commit, or if a rebase goes wrong



Bisect

- Used to find the exact commit where a bug was introduced
- Start by specifying last known good commit
- Bisect will checkout commits in between and ask if they're good or bad
- At the end, bisect knows exactly which commit was the first bad one

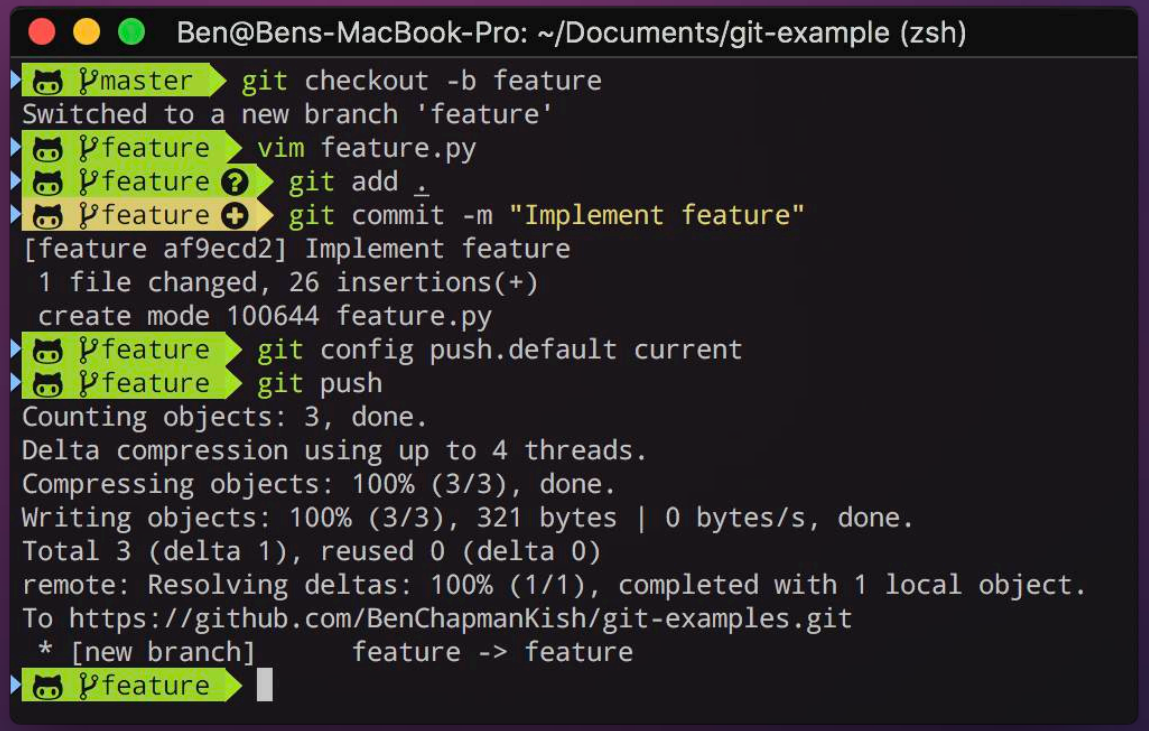


Tips and tricks

to make your life easier

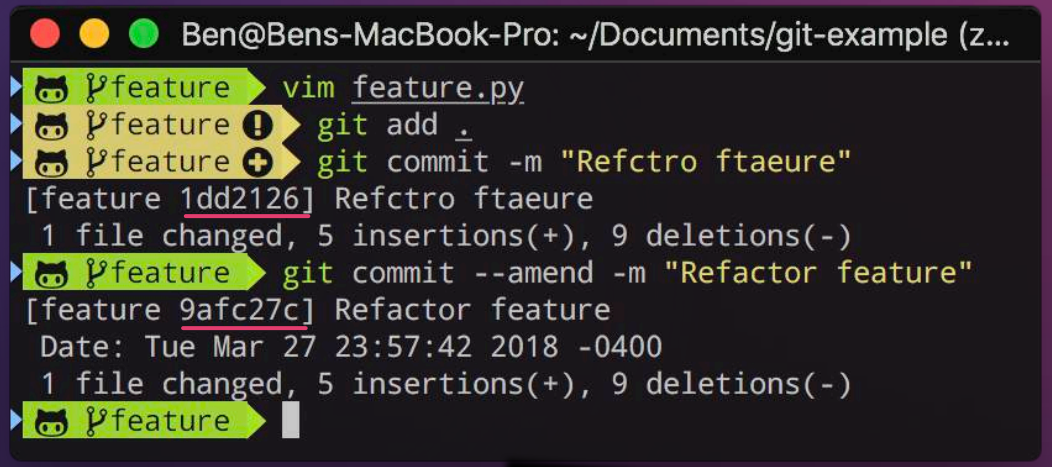
Pushing

- Set your default push action to current
- `git config push.default current`

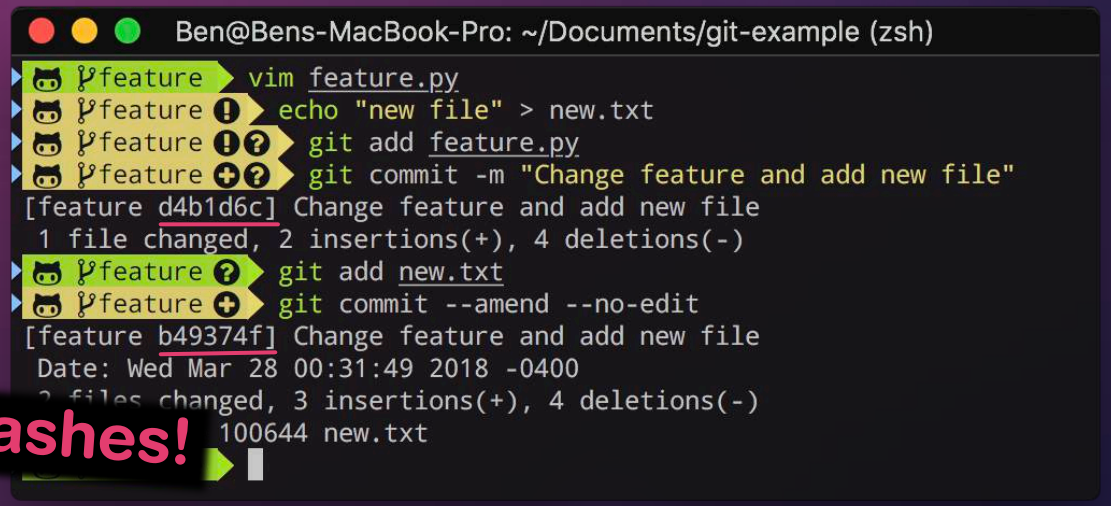


Amending commits

- Fix typo in commit message
- Add file you forgot to stage
- `git commit --amend`



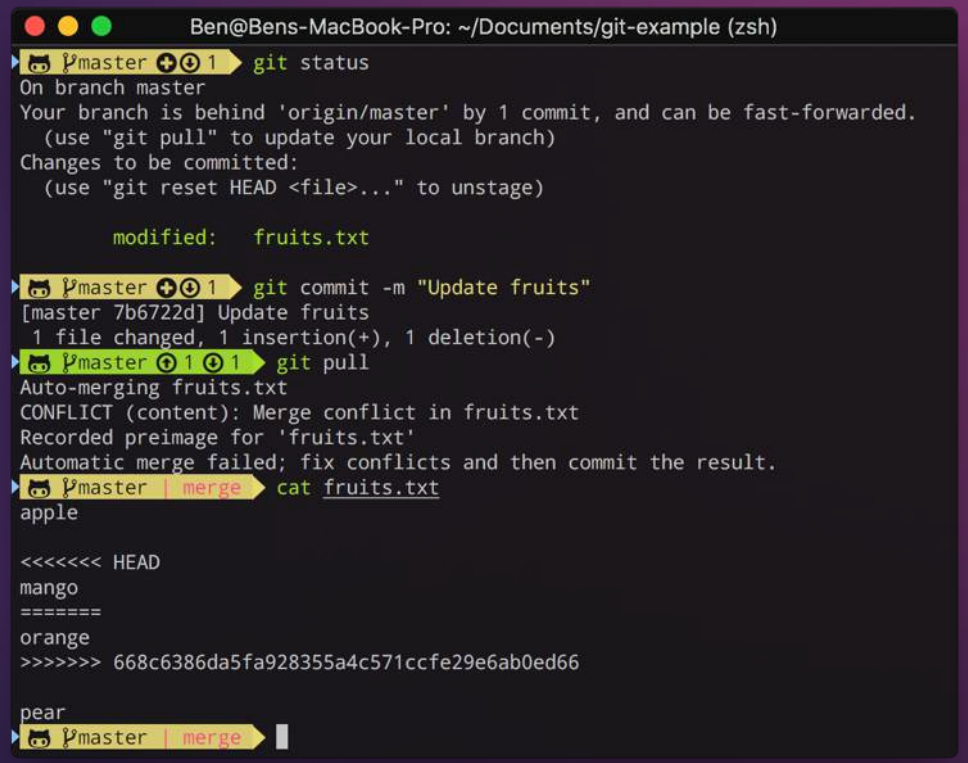
- Remember: commits can't truly be changed, this actually makes a new commit with your changes
- If you already pushed, you'll have to force push

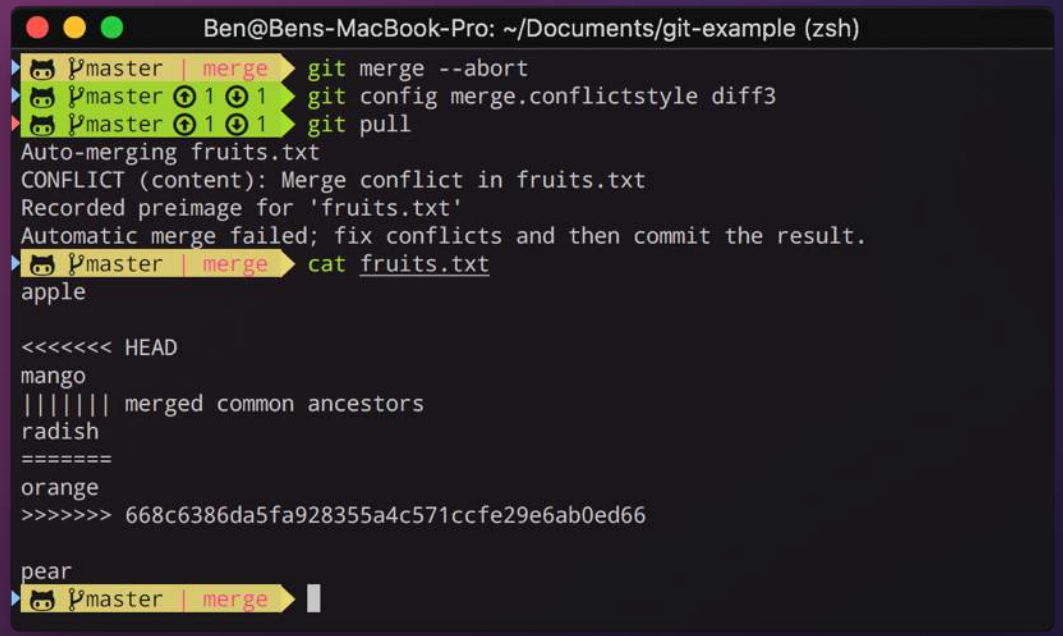


Note the different hashes!

Merge conflicts

- Set your merge conflict style to diff3
- `git config --global merge.conflictstyle diff3`





Merge conflicts

- If someone else indented a bunch of lines and it's causing lots of conflicts
- `git merge feature -Xignore-all-space`

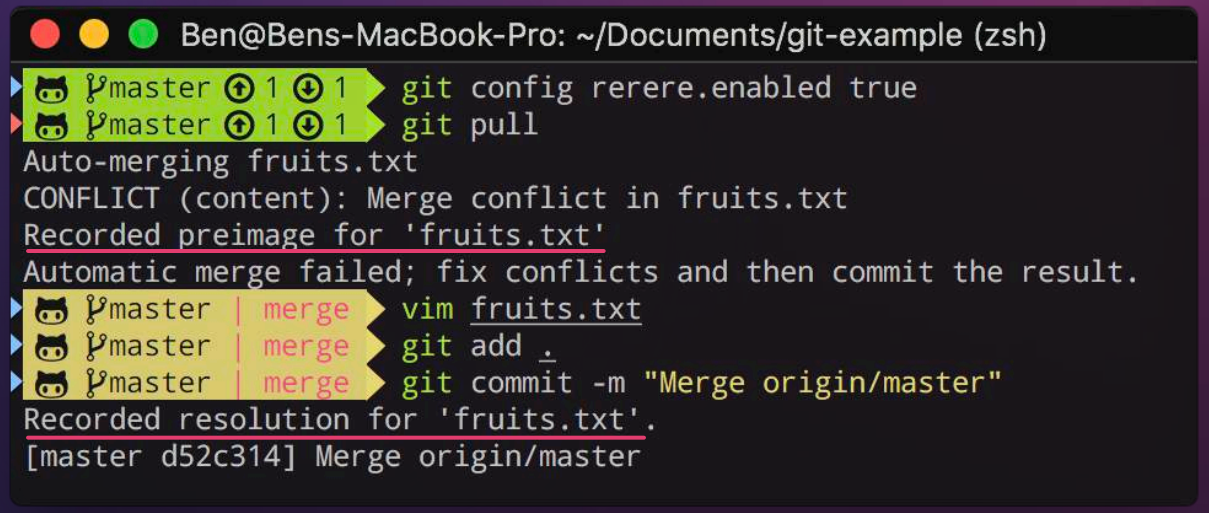
Diff algorithms

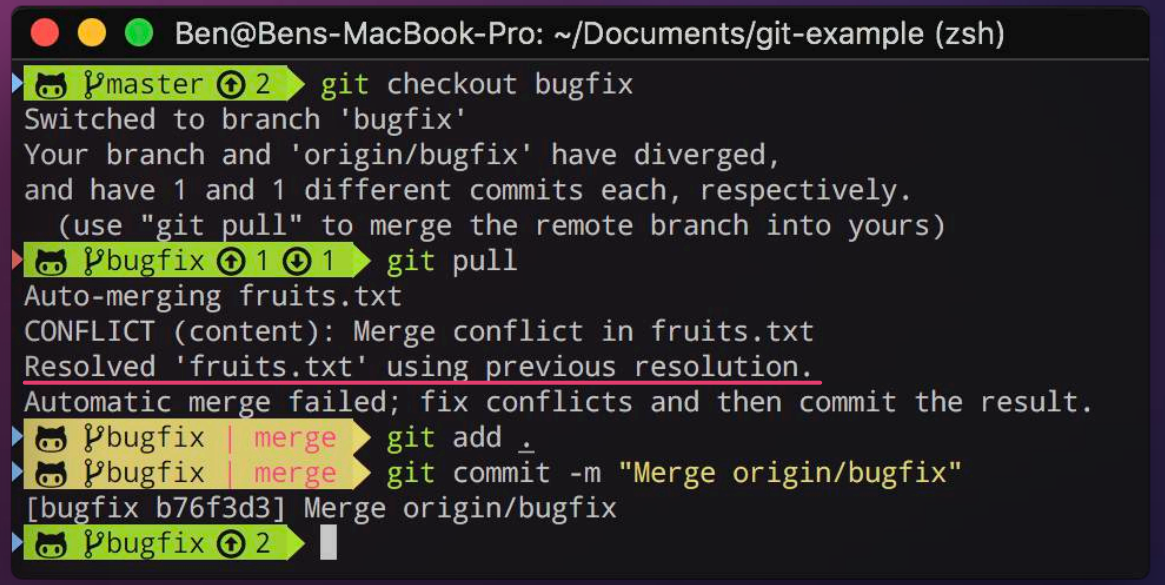
- Some diff algorithms can make much more sense than others
- `git diff --diff-algorithm=patience`

- Can also ignore whitespace changes with diff too
- `git diff -w`

Reuse recorded resolutions

- Never fix the same merge conflict twice!
- `git config --global rerere.enabled true`





Handy shortcuts

- Checkout the previous branch you were on: `git checkout -`
- Reset *n* commits back on the current branch: `git reset @~n`
- Add files and commit at the same time `git commit –a –m “message”`

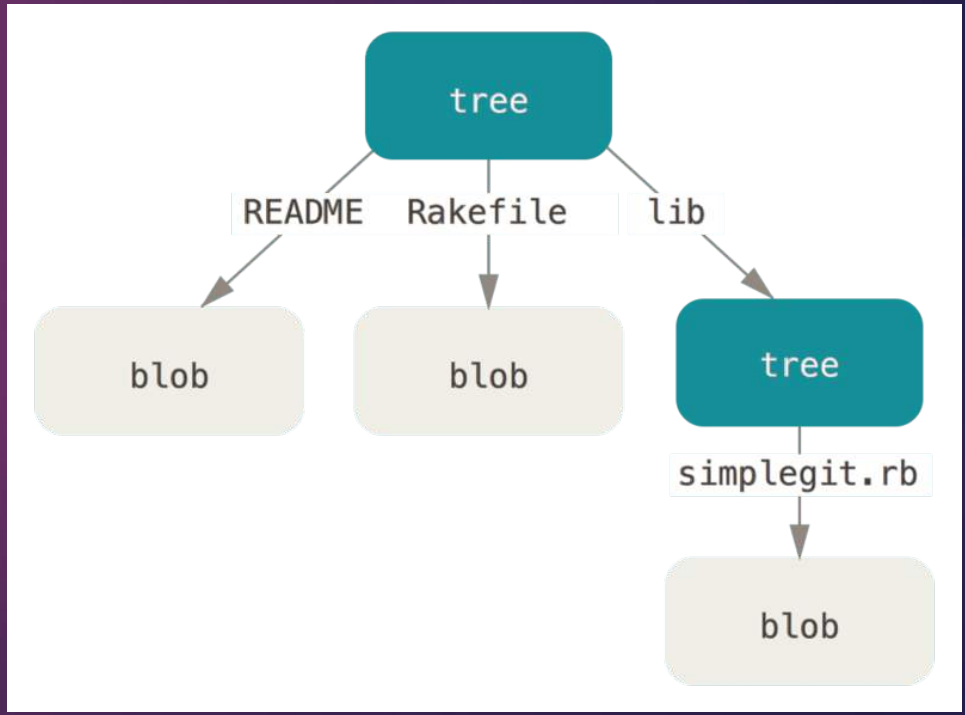
How does Git *really* work?

How Git *really* works

- Git is really a *content-addressable filesystem* with a VCS interface written on top of it
- Internally, Git has a key-value store of objects and their SHA-1 hashes (the hash is the key and the object is the value)
- These objects, among other Git internals, are stored in the `.git` directory at the root of every Git-controlled project

Git objects

- There are several types of objects that Git stores:
  - *Blob:* content; text/code/images/etc.
  - *Tree:* a collection of pointers to blobs and other trees, and names for each of these
  - *Commit:* A pointer to a tree, with metadata such as parent commits and a commit message

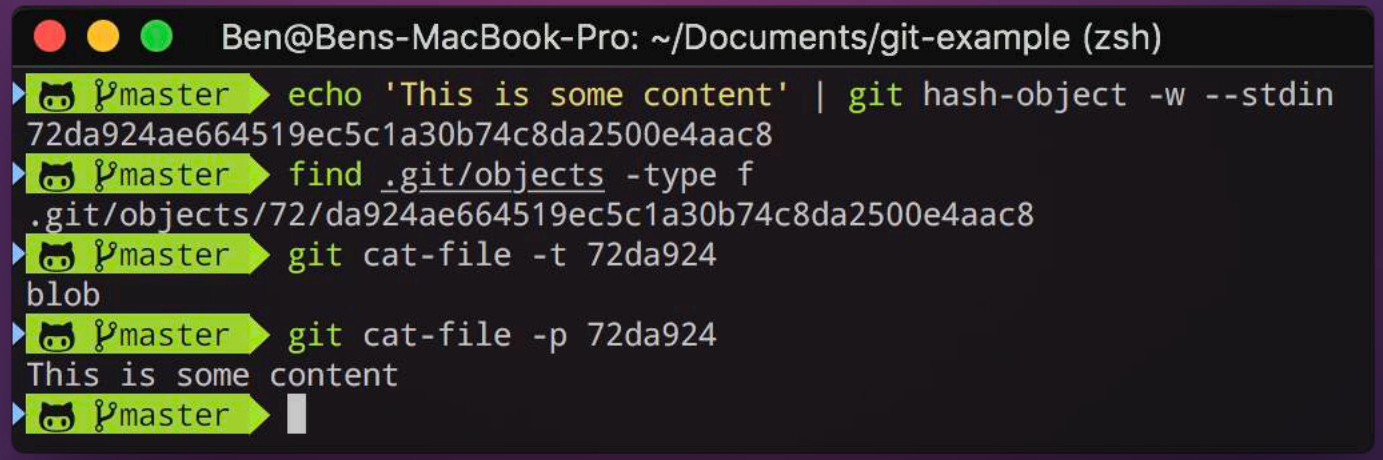


Plumbing and porcelain

- There are two kinds of Git commands:
  - The commands we use every day are called *porcelain* commands
  - Each of these actually uses low-level Git commands called *plumbing* commands
- Let's try using plumbing commands to do some basic Git operations!

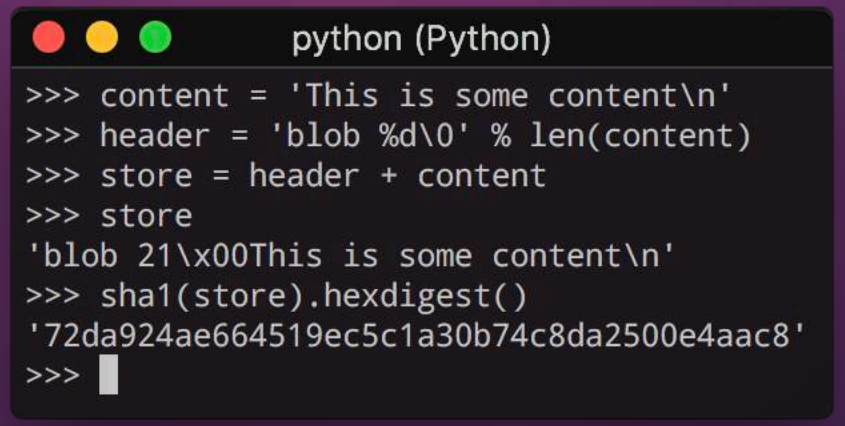
Creating a blob

- Hashing content and storing the blob in the objects database
- No filename?



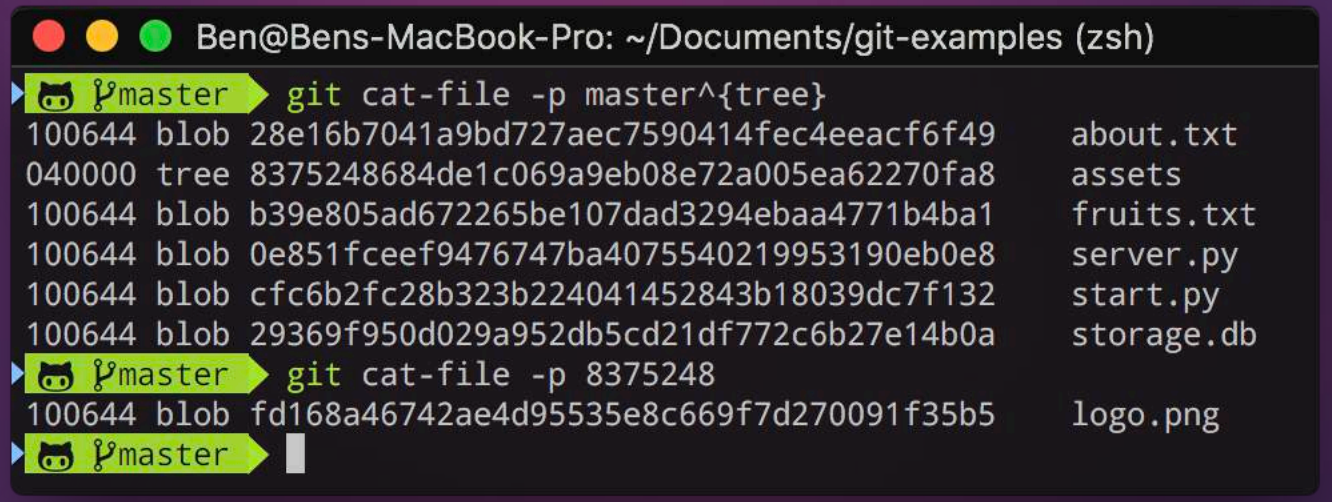
Hashing the blob

- To get the hash, Git doesn't just hash the content
- It also prepends a header

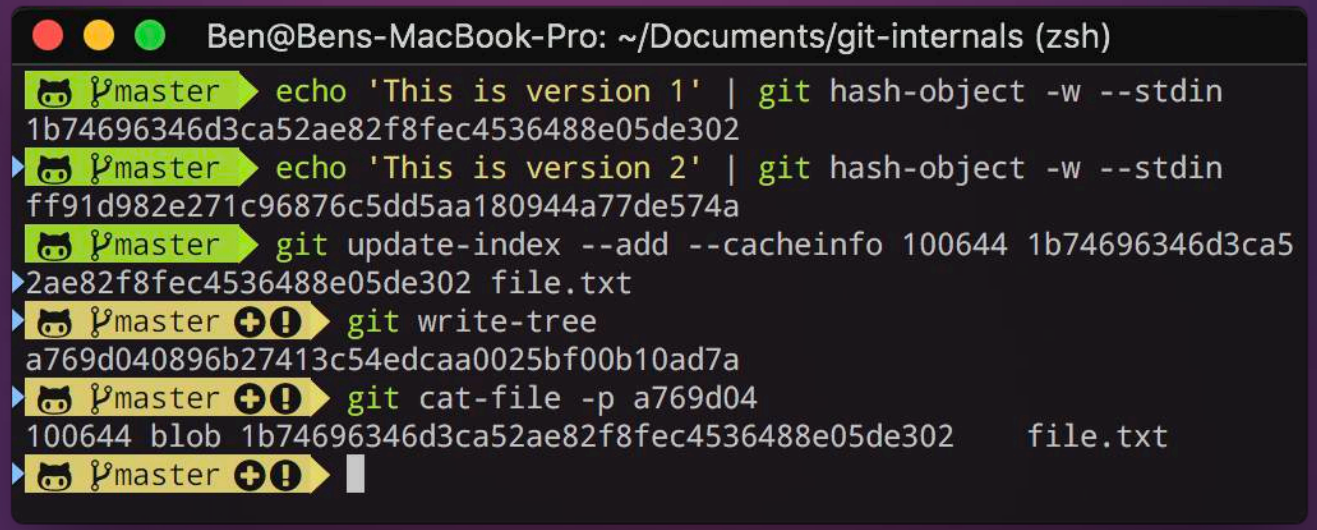


Reading a tree

- The tree stored in a commit on an actual project may look like this:



Creating a tree

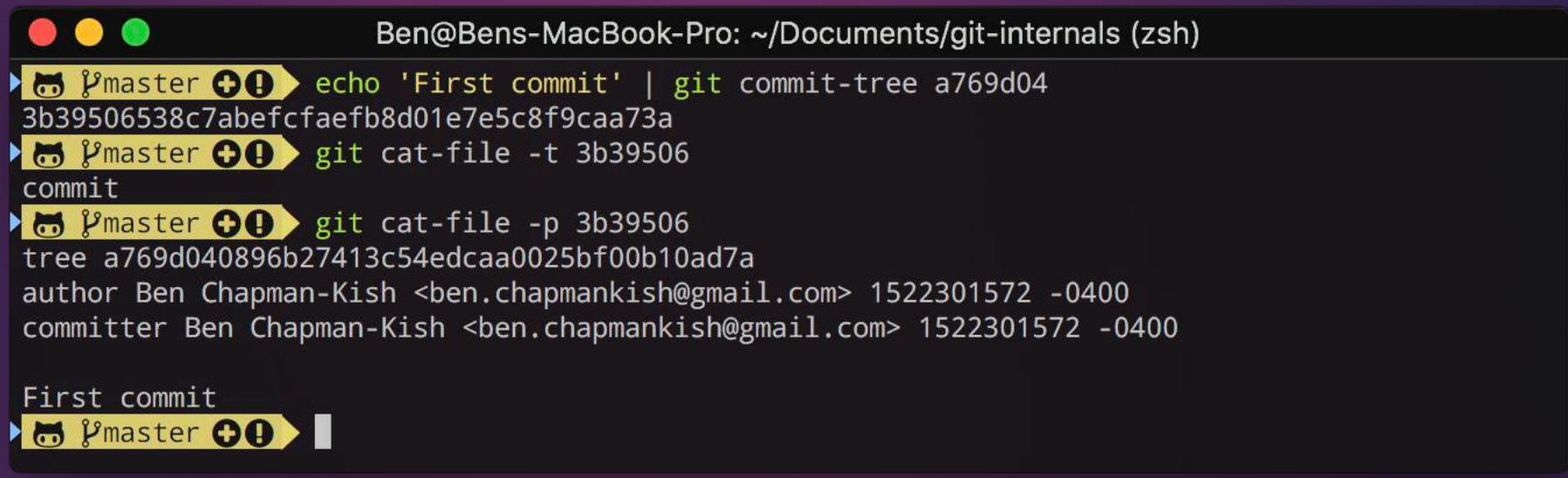


Creating another tree

```
Ben@Bens-MacBook-Pro: ~/Documents/git-internals (zsh)
 master +!  echo 'This is a new file' > new.txt
 master +!?  git update-index --add --cacheinfo 100644 ff91d982e
271c96876c5dd5aa180944a77de574a file.txt
 master +!?  git update-index --add new.txt
 master +!  git write-tree
9642c5ff94d0aae6aa40144c6e8c259d9e12a7b6
 master +!  git cat-file -p 9642c5f
100644 blob ff91d982e271c96876c5dd5aa180944a77de574a    file.txt
100644 blob 6dfa057f0d4a43d5a3025a9c14dea607de9e1dbb    new.txt
 master +!
```

```
Ben@Bens-MacBook-Pro: ~/Documents/git-internals (zsh)
 master +!  git read-tree --prefix=subdirectory 9642c5ff94d0aae6aa40144c6e8c259d9e12a7b6
 master +!  git write-tree
923bc6ec9703fa92c24d12037373ec38cf98803b
 master +!  git cat-file -p 923bc6e
100644 blob ff91d982e271c96876c5dd5aa180944a77de574a    file.txt
100644 blob 6dfa057f0d4a43d5a3025a9c14dea607de9e1dbb    new.txt
040000 tree 9642c5ff94d0aae6aa40144c6e8c259d9e12a7b6    subdirectory
 master +!
```

Creating a commit



Creating a child commit

Ben@Bens-MacBook-Pro: ~/Documents/git-internals (zsh)

```
master ➜ echo 'Second commit' | git commit-tree 923bc6e -p 3b39506
72f6844e48761ffb17e2993079d2aa3855e312ad
master ➜ git cat-file -p 72f6844
tree 923bc6ec9703fa92c24d12037373ec38cf98803b
parent 3b39506538c7abefcfaefb8d01e7e5c8f9caa73a
author Ben Chapman-Kish <ben.chapmankish@gmail.com> 1522301752 -0400
committer Ben Chapman-Kish <ben.chapmankish@gmail.com> 1522301752 -0400

Second commit
master ➜ git log --stat 72f6844
```

git log --stat 72f6844 (less)

```
commit 72f6844e48761ffb17e2993079d2aa3855e312ad
Author: Ben Chapman-Kish <ben.chapmankish@gmail.com>
Date:   Thu Mar 29 01:35:52 2018 -0400

    Second commit

 file.txt              | 2 +-
 new.txt               | 1 +
 subdirectory/file.txt | 1 +
 subdirectory/new.txt  | 1 +
 4 files changed, 4 insertions(+), 1 deletion(-)

commit 3b39506538c7abefcfaefb8d01e7e5c8f9caa73a
Author: Ben Chapman-Kish <ben.chapmankish@gmail.com>
Date:   Thu Mar 29 01:32:52 2018 -0400

    First commit

 file.txt | 1 +
 1 file changed, 1 insertion(+)
(END)
```

There's so much more!

- Submodules
- References
- Packfiles
- Transfer protocols
- Garbage collection
- The refspec
- Git hooks

That’s all for today!