Github Webinar - 2012.01.10 Documentation

Release 0.1

Evan Culver

October 21, 2016

Contents

CHAPTER 1

Session 1 (8:00a - 8:50a)

Introduction, a little history.

git init <projectname>

.git folder contains entire commit history

user identity - separate from auth, merely who you are.

locally, there are no permissions present other than those of the filesystem itself.

git config –local = per repository –global = per user –system = per machine

color.ui auto

core.autocrlf input – "This setting tells git to convert the newlines to the system's standard when checking out files, and to LF newlines when committing in"

ssh-keygen -t rsa -C"<email> for <purpose>"

-C = comment

CHAPTER 2

Session 2 (9:00a - 9:50a)

Committing code

Files being bumped out of stage (editing a file once it's staged)? Not really. What really happens? Why is the file shown in unstaged and staged? There are two sets of changes. Two scopes, file and content. Git separates them in to two sets of changes – to see the difference:

`git diff`

compare to

`git diff --staged`

to see what's going on.

`git commit -v`

Interatcive add:

```
``git add -p``
```

Filtered Diff:

```
``git diff -diff-filter=M``
```

A = Added C = Copied D = Deleted M = Modified R = Renamed T = Type changed U = Unmerged X = Unknown B = Broken pairing

Normal `git diff` – highlights whitespace changes.

`git diff -w` – surpresses whitespace changes.

`git diff --color-words HEAD`

`git add -A` – adds any new files whereas `git add .` will just add modified.

CHAPTER 3

Session 3 (10:00a - 10:50a)

Viewing history

3.1 Git log

No options: reverse chronological order. Automatically piped to pager.

Rarely used with no options.

`git log --stat` – shows files that participted in each commit. Commit hash, author, date, files, +/- for lines per change.

Extremely tedius to page our way back through changes.

Milestone marker for class: if we stopped now, you should have a pretty good basic workflow for getting around day-to-day.

More advanced usage:

```
``git log --diff-filter=A`` -- Same filters as ``git diff``
```

`git log --pretty=raw` – raw, full, fuller, and even custom with sprintf

`git log -3` – limits number of commits to 3 in this case.

`git log --author=evan@wiredrive.com` – all commits authored by me.

`git log --author=Evan` – all commits by me by using my name.

3.2 Ignoring Files

Flat text file called .gitignore

Per project: in project root

Example:

```
*.log
*.tmp
```

Ignores all files with .log and .tmp exensions.

Note `git status -u` – show me all files, don't simplify, lists out all files below folders.

Applies, at evaluation time in memory.

You can add .gitignore files to subfolders, behave like .gitconfig:

Top has weakest precidence, bottom has highest precidence.

They're evaluated as if concatenated end-to-end.

.gitignore can exist in any number of directories and subdirectories with no limit.

Git does not track empty directories.

How do you track empty directories? "holding it open with a toothpick."

Usually the easiest way is to touch a .gitignore in that directory.

Preconfigured .gitignore files: https://github.com/github/gitignore

System-wide?

```
git config --global core.excudesfile "~/.gitignore"
```

Have to enable, can't just create the file.

3.3 Removing Files

Closure to the add command

Basis for refactoring and renaming

`rm <filename>`

`git rm <filename>` – put deletion into staging, if file doensn't exist, just adds deletion to staging area

***example*::** rm <files> git add -u .

`git add -u` grabs all removed files.

`git rm <fileame>` removes the file and stages that file.

If you want it back:

`git reset --hard` – we'll come back to this, but it returns everything that is tracked to their last committed state. Blows away everything that's staged.

note: untracked files are untouched.

There is no `move` primitive in git.

`git mv <src> <dst>` – a 'composed' command for convenience.

Same as:

```
``mv <src> <dst> ; git add <dst>``
```

Rename flagged as soon as file is re-added. Git is good at detecting similar content.

There's a pretty funny discussion on the git mailinglist as to why there is no move primitive. Might be worth a look. (See slides).

`git log --stat -M` – shows renames in history.

Mindset should be that no matter how accurate we are when committing to history, there's always a chance that the committer has been mistaken at commit time. Just move it to the display layer.

Q: Can you set a threshold on the similarity filter? A: Yes, `git log --stat -M90`.

`git log -2 --stat -C --find-copies-harder` – Tells it to look harder for copy detection on top of rename detection.

This stuff can be used for the simplification of merges and is relevant for refactoring.

CHAPTER 4

Session 4 (12:00p - 2:00p)

4.1 Questions

- Can you reset an individual file?

```
git checkout -- <filename>
```

```
git checkout HEAD -- source/
```

What's the difference between

```
git checkout -- <filename>
```

and

```
git checkout HEAD -- <filename>
```

Nothing, HEAD is assumed.

- Other questions on collaboration, binaries and design...

4.2 Netork Interactions

Everything begins locally in git then layers on things like remotes and collaboration.

Almost all activities happen offline. Offline activities have to be pushed to remotes.

4.2.1 Cloning

four primary protocols: file, git, ssh, http.

4.2.2 file

```
git clone file:///path/to/project
```

```
git clone /path/to/project
```

cloning a local file is like checkout, but clone grabs all history for a repository.

file:///path/to/project fires up the 'networking' code within git whereas just /path/to/project looks directly at the filesystem.

4.2.3 git

```
git clone git://server/project.git
```

Almost always read-only. Can be made writable, but in most cases it's read-only.

4.2.4 http (dumb)

```
git clone http://server/project.git
```

```
git clone https://server/project.git
```

Used to be the real differntiator between git and mercurial. Dumb version is insanely slow and inefficient.

4.2.5 https (smart)

Much faster and offers progress. Mostly handled on the server side.

4.3 Exercises

Cloning and collaborating:

```
git clone https://githubstudents:students987@github.com/githubstudents/hellogitworld.git
```

```
git branch evanculver
```

`git push -u` – tells git to track it.

<BREAK>

Example of git's speed by adding 3000 files and pushing them to remote.

After pushing, we can almost instantly get references to those files, but they're just references.

`git fetch origin` `git checkout thousands`

When checking out, the state of the filesystem is being modified. By switching between the `thousands` branch and mine, > 3000 files are being created/destroyed

"Once mastered, the concepts of local, remote and upstream branches, the rest falls into place"

Local and remote live on your box.

Typing `git fetch`, your local cache is updated against what's upstream or remote to you.

`git fetch` === `git fetch origin`

4.3.1 Push

```
git push <remote>
```

4.3.2 Pull

0 collaboration up until this point. The integration point.

Combination: retrieves upstream objects and then merges and then commits merge to the local branch.

4.3.3 Excercise: Merging in remotes

Merging:

```
git merge remotes/origin/chance
```

Pushing back up to remote:

```
git push
```

Reviewing any commits in the entire remotes tree:

```
git log -2 remotes/origin/chance
```

4.3.4 Remotes

Simiply 'bookmarks' or pointers.

Origin is a remote you get by default.

```
git remote -v
```

Q: Why two entries? A: One inbound and one outbound. Almost always the same, but can be different in a very unique situation. For example, you may want to pull all code over ssh but push all over http.

4.3.5 Adding a remote

```
git remote add <remotename> <remotepath>
```

```
git remote add matthewongithub https://github.com/matthewmccullough/hellogitworld
```

note: Remote branches are locally immutable.

We have to make a local branch for anything we would like to change.

```
git checkout kenkil
```

Git is smart enough to know that you probably wanted to actually checkout the remote (`remotes/origin/kenkil`).

If you look at `.git/config`, you will see various `[branch *]` sections which define the remote/tracking portion of this.

Git will get mad if the branch name is ambiguous:

error: refspec... blah blah

```
git checkout -b <localbranch> --track remotes/origin/richzurad
```

4.3.6 Prune

`git remote prune <remotename>` – only removes old branches that no longer exist on the remote, has to be given an explicit remote name.

4.3.7 Diff'ing against remotes (or any two things)

```
git diff origin/ericg
```

```
git diff origin/ericg origin/richzurad
```

4.3.8 More exercises

<edit, add, commit>

`git reset --soft HEAD~1`

Tells git to roll back 1 commit, leaving the change in tact.

If you have already pushed, the game has changed entirely.

You can do the same as above, but force the push – the consequences of forcing the push causes the graph of the commits changes as your peers see them. Furthermore, you then have to ask what other people may be basing their work on. It may cause some really counter-intuitive output and attempts to merge in changes.

A better approach is to push another commit by 'reverting':

`git revert <commithash>` – Make a new commit that negates what was just committed. Always safe because it's always forward moving which is exactly why forcing commits and changing history/graphi is usually always bad.

CHAPTER 5

Session 5 (2:15p - 2:15p)

5.1 Creating a funky situation

`git checkout -b evanculver2`

<edit a particular line>

<add, commit>

`git checkout -b evanculver`

<edit same line as before>

<add, commit>

`git merge evanculver2`

EEEEK –> merge conflict

“Make the file appear as a logical combination of those two disparate pieces”

5.2 Pull requests and workflow

Pull requests aren’t limited to cross-repository merges. Pull requests make perfect sense within a single repository for forced code-review.

5.3 Rebase

Simulates members taking turns working? Actually not stupid, just sounds ridiculous.

Branching: you usually wait as long as possible to branch, and you try to get your changes merged back in as soon as possible (because your VCS sucks).

Rebasing rewrites history, keeps a consistent, straight history.

Merge commits become integration commits by layering merges on to the end via fast-forward (instead of merge made by recursive).

Fast-forward means that only one of the two branches changed since the branch occurred.

No merge commits are created.

`git checkout feature/some-feature` `git rebase master`

"Take all of the stuff in the feature branch, hibernate it, delete it, start it at the end of master then replay all of the work in the feature branch on master."

"Rework your work so that it makes more sense for your team."

```
git checkout feature/some-feature git rebase -i HEAD~5
```

"Rebase seems like a tool to replay history."

Gist created by Matt with his notes:

https://gist.github.com/gists/1589518

CHAPTER 6

Indices and tables

- genindex
- search